ESD-TR-86-282 3285-4-253/1

Program Office Guide to Ada Edition I

C.N. AUSNIT
E.R. ANSAROV
N.H. COHEN

AFGL/SULL
Research Library
Hanscom AFB, MA 01731

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254

17 September 1986

Prepared For

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR DEVELOPMENT PLANS
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731

ADA183226

LEGAL NOTICE

When U.S. Government drawings, specificotions or other dato ore used for any purpose other than a definitely reloted government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; ond the foct that the government moy hove formulated, furnished, or in ony way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise os in ony manner licensing the holder or any other person or conveying ony rights or permission to manufocture, use, or sell any patented invention thot may in ony way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

MARK V. ZIEMBA, 2Lt, USAF
Project Officer, Software
Engineering Tools & Methods

ARTHUR G. DECELLES, Capt, USAF
Program Manager, Computer Resource
Management Technology (PE 64740F)

FOR THE COMMANDER

ROBERT J. KENT
Director
Software Design Center
Deputy for Development Plans
and Support Systems

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 3285-4-253/1	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-86-282
6a. NAME OF PERFORMING ORGANIZATION SofTech, Inc. — 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Hq, Electronic Systems Division (XRSE)
6c. ADDRESS (*City, State, and ZIP Code*) 460 Totten Pond Road Waltham, MA 02254	7b. ADDRESS (*City, State, and ZIP Code*) Hanscom AFB Massachusetts, 01731
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Deputy for Development Plans — 8b. OFFICE SYMBOL (*If applicable*) ESD/XRSE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33600-84-D-0280
8c. ADDRESS (*City, State, and ZIP Code*) Hanscom AFB Massachusetts, 01731	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO — PROJECT NO — TASK NO — WORK UNIT ACCESSION NO
11. TITLE (*Include Security Classification*) Program Office Guide to Ada Edition I	
12. PERSONAL AUTHOR(S) C.N. Ausnit, E.R. Ansarov, N.H. Cohen	
13a. TYPE OF REPORT Technical — 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (*Year, Month, Day*) 1986 September 17 — 15. PAGE COUNT
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB-GROUP	18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*) Ada, Ada Compiler, AJPO, Run-Time Support, Computer Languages

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The purpose of the Program Office Guide to Ada is to discuss issues affecting the selection development and maintenance of systems whose software is written in the Ada language. Each volume focuses on a different set of topics and their implications for managers.
This edition concentrates on: Policy, run-time efficiency, customization of run-time support environments, training, Ada program design languages and conversion to non-Ada code.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL M.V. Ziemba, 2Lt, USAF	22b. TELEPHONE (*Include Area Code*) (617) 377-2656 — 22c. OFFICE SYMBOL ESD/XRSE

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

ACKNOWLEDGEMENTS

This report is sponsored by the Computer Software Systems Program Office, Software Design Center (XRSE), Electronic Systems Division (ESD), United States Air Force Systems Command, Hanscom AFB, Massachusetts 01731. Funding for the effort was provided by the Air Force Computer Resource Management Technology Program, PE 64740F, Project 2526 - Software Engineering Tools and Methods.

Program Element 64740F is the Air Force engineering development program to develop and transfer into active use the technology, tools, and techniques needed to cope with the explosive growth in Air Force systems that use computer resources. The goals of the program are to: (a) provide for the transition of computer system developments in laboratories, industry, and academia to Air Force systems; (b) develop and apply software acquisition management techniques to reduce life-cycle costs; (c) provide improved software design tools; (d) address the various problems associated with computer security; (e) develop advanced software engineering tools, techniques, and systems; (f) support the implementation of high-order languages, e.g. Ada; (g) address human engineering for computer systems; and (h) develop and apply computer simulation techniques for the acquisition process.

TABLE OF CONTENTS

SECTION 1

EXECUTIVE SUMMARY

This report is the first of four editions for the Program Office Guide to Ada*. It complements the Program Manager's Guide to Ada, ESD-TR-85-159, dated May 1985. This effort was sponsored by the Air Force Computer Resource Management Technology Program, Program Element 64740F, Project 2526, Software Engineering Tools and Methods.

The purpose of the Program Office Guide to Ada is to discuss issues affecting the selection, development, and maintenance of systems whose software is written in the Ada language. Each edition focuses on a different set of topics and their implications for managers. Points of contact are provided where available.

The first edition concentrates on:

- policy
- run-time efficiency
- customization of run-time support environments
- training
- Ada Program Design Languages
- conversion of non-Ada code

The Air Force is continuing its policy that software developed for major systems and Air Force designated acquisition programs must be written in the Ada programming language.

For all other programs a new version of AFR 800-14 has been drafted which requires the Computer Resource Working Group (CRWG) to consider the use of Ada and make a formal recommendation to the program manager. Only DOD-validated Ada compilers may be used and only in accordance with the procedures for risk management for programming languages in attachment 4 to AFR 800-14.

Waivers require solid technical and programmatic justification. Waiver approval authority is now coordinated at a much higher level than initially, specifically between the offices of HQ USAF/SC, HQ USAF/LE and HQ USAF/RD.

---

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

The Ada Joint Program Office is coordinating a draft document explaining validation policies and procedures. Based on three years practical experience, these proposed new validation policies and procedures provide for more validated Ada compilers, available sooner. This decreases the risks associated with Ada compiler availability. Program managers can minimize their risks for developing standard Ada code by selecting for a project's baseline compiler which has passed all of the Ada compiler Validation Capability tests as recently as possible.

Production quality compilers along with benchmarks to measure their efficiency are appearing in the marketplace. There are several optimizations that a programmer can specify to improve code performance further. In addition, there are several alternatives in custom-tailoring the run-time support environment to achieve maximum performance.

The Ada Joint Program Office has launched an initiative to promote tool portability through the Common APSE (Ada Programming Support Environment) Interface Set. A draft DOD MIL-Standard is in final coordination and several prototypes already exist.

The successful transition to Ada will require substantial training and retraining of the work force. This training should cover environment and methodology in addition to the language itself. Managers, programmers and support personnel will need varying degrees of training. There are several efforts underway studying formal ways of measuring the effectiveness of training.

Another aspect of the Ada transition is whether or not existing code should be translated into Ada. Given the current technology, using a multi-language interface mechanism, if supported by the run-time system, is preferable to converting code whose design may be obsolete and poorly documented.

Ada provides an ideal base for a Program Design Language. Several government and industry organizations have developed Ada-based PDLs, and an IEEE Working Group has drafted a Recommended Practice. An important consideration is whether a compiler is a sufficient PDL processing tool or a more sophisticated, specialized tool is needed.

SECTION 2

POLICY UPDATE

This section describes current Air Force and Ada Joint Program Office (AJPO) policy with regard to the use and implementation of Ada. The original policy documents that were provided in Appendix 2 of the original Program Manager's Guide to Ada are still valid; they are, therefore, not reproduced in this Edition.

2.1 Air Force Policy

Section 2.1.1 reviews the Air Force policy on the introduction of Ada. Section 2.1.2 discusses the current position regarding waivers.

2.1.1 Ada Introduction Plan

The Air Force continues to follow its Interim Policy on Computer Programming Languages that software developed for major systems and Air Force designated acquisition programs must be written in the Ada programming language. Certain major systems have been designated to be Defense Systems Acquisition Review Council (DSARC) or Air Force Designated Acquisition Program (AFDAP). Typically these systems have research, development, test and evaluation costs that exceed $100 million, have production costs that exceed $500 million, or have special interest for the Department of the Air Force or the Secretary of Defense. These acquisition programs have been mandated to use Ada. An update on their status may be found in Section 2.3. For all non-major programs, the upcoming new version of AFR 800-14 will require that the CRWG consider the use of Ada and make a formal recommendation to the program manager.

All major mission critical programs with a scheduled Milestone I review after 1 Jan 84 or a scheduled Milestone II review after 1 July 84 must use Ada or seek a waiver. Mission critical programs are defined to include:

- Intelligence systems
- Cryptologic systems related to national security
- Command and control of military forces
- Integral parts of weapons systems
- Systems critical to the direct fulfillment of military or intelligence missions (e.g. logistics, planning, environmental, warning, etc.)

Unless a program has made a formal language commitment prior to 10 Jun 83, any mission critical program that wishes to use a different language must request a waiver, discussed in Section 2.1.2 below. In other words, there will be no retroactive Ada designations. All programs, mission and non-mission critical, are strongly encouraged to use Ada. Ada is voluntary for use on automated data processing (ADP) systems. It is an approved ADP language because it is now a Federal Information Processing (FIPS) standard.

The four phases of the Air Force Systems Command (AFSC) Ada introduction plan represent logical rather than chronological phases, and there are funded programs distributed across the first three phases. Phase I consists of the use of Ada in the lab, as in the Wright Patterson AFB AFWAL Ada Based Integrated Control System (ABICS) F-15 program. Phase II, the dual development phase is characterized by such programs as the Common Ada Missile Package (CAMP) program by AFATL at Eglin AFB and the Mobile Information Management System (MIMS) for SAC at Offutt AFB. Two programs, MILSTAR and WIS, are currently in Phase III, the selected use phase. MILSTAR has selected Ada as the major implementation language for the Ground Control Segment. The WIS baseline design uses Ada, and a formal language decision is expected by December 1986.

A Regulation 800 series is under development. This draft regulation, 800-14, will direct that the CRWG, comprised of the using, developing, operating, and maintaining commands, work out all of the life cycle arrangements for support of computer resources. These arrangements must consider the use or nonselection of Ada as a PDL and/or an implementation language. The net effect is that because every program has a CRWG, every program will have to address the use of Ada. This regulation will not affect the required use of Ada on major programs.

Within Air Force Systems Command if Ada is not used for on-board avionics software in aircraft, tactical & strategic missiles, munitions or space systems then this software must be written in JOVIAL J73. Only Air Force validated JOVIAL J73 compilers may be used and only in accordance with the procedures for risk management for programming languages in attachment 4 to AFR 800-14.

2.1.2 Waiver Policy

The Air Force waiver policy continues to follow the 13 January 1984 memo. (This memo is reproduced in Section 3.6 of the original Program Manager's Guide to Ada.) Ada waiver approval is now at a higher level than initially, through HQ

USAF/SC. Furthermore, Ada waivers must be coordinated through the offices AF/LE and AF/RD.

Waivers require solid technical and programmatic justification. They must be thoroughly documented and submitted in a timely manner, as early as possible in the system development life cycle so as not to be overtaken by events, for instance as part of the source selection process. Furthermore, waivers should not be written for an entire program but only for as restricted a part of the system as possible, in other words for a specific time/space critical and hardware dependent component. Additional information that should be filed with a waiver application includes program name, overall description, and phase (source selection, full-scale development, etc).

The technical justification for a waiver must, wherever possible, provide hard data on the timing, memory and technical capabilities (such as interrupt handling, extended addressing ability, etc). Information such as benchmark results, technical references and source to code expansion ratios should be provided. Waiver requests should also describe language alternatives considered, including technical data showing that the alternate language is able to meet requirements where Ada is unable to do so. For example, efficiency alone is not a sufficient reason not to use Ada if efficiency is not a critical factor.

A waiver request must include a life cycle cost analysis. Factors to consider are the extent of modifications to an existing system, the availability of commercial off the shelf (COTS) software, the existence of a vendor maintenance organization, cost estimation models, training and compiler costs, and productivity effect. In addition to the life cycle cost analysis, the waiver application should discuss the impact on training, especially the training for personnel who must provide long term system support.

The Computer Resource Focal Point (CRFP) provides the first level of review in the waiver process. Appendix 1 contains a list of the CRFP's at the product divisions and labs at the time of this writing. The CRFP forwards the application with his or her recommendations to the developing Air Force Major Command (such as AFSC, TAC, SAC, etc). A recommendation to disallow a waiver carries a great deal of weight; moreover, the lack of a recommendation on a waiver raises serious questions at higher review levels. Systems Command works with Air Staff for final disposition of waivers.

2.2 Ada Joint Program Office (AJPO)

The Department of Defense continues to show a high-level interest in and commitment to the use of Ada. Section 2.2.1 summarizes the status of forthcoming DoD Directives. Validation policy is reviewed in Section 2.2.2 below. The effect of this policy on risks and risk management is deferred to Section 3.

2.2.1 DoD Directive Status

Dr. Donald A. Hicks has issued a memorandum dated 2 December 1985, stating the Department of Defense's continued commitment to Ada. This memo, reproduced in Appendix 2, reaffirms Dr. Delauer's 10 June 1983 memo specifying the Ada language as "the single common, computer programming language for Defense mission-critical applications."

Draft DoD Directive 5000.29 and Draft DoD Instruction 5000.31 are in the review cycle. A copy of the draft of 5000.29 may be found in Appendix 1 of the Program Office Guide to Ada.

2.2.2 Validation Policy

Validation is the process through which a compiler is demonstrated to be in compliance with the MIL-STD-1815A Ada language standard. A compiler must successfully pass all of the Ada Compiler Validation Capability (ACVC) tests in order to be designated an Ada compiler and to receive a validation certificate. A validation certificate is valid for one year and applies only to the base host/target compiler configuration tested under a specific ACVC suite. At the time of this writing, Ada compilers are tested against the ACVC Suite Version 1.8, comprising some 2700 tests. Validation in and of itself does not guarantee a production quality compiler, and further acceptance testing should be conducted. Sections 3 and 6 of this document provide further guidance on the implications of validation policy as well as on compiler and environment evaluation.

The AJPO has issued a draft version of the Ada Validation Policies and Procedures Document, dated 10 February 1986. This document describes the definitions, policies and procedures for Ada compiler validation. It is divided into three parts, the first two of which are reproduced in Appendix 3. Parts I and II discuss the general framework and policies of validation. Part III, available through the AJPO, enumerates the specific validation procedures.

Key terms in the Ada Validation Policies and Procedures Document are the concepts of a derived compiler and a project-validated compiler. A derived compiler is a compiler which is a slightly modified version of a validated compiler. The modifications would typically reflect the changes needed to accommodate a family of target architectures. A derived compiler may be registered with the AJPO and considered as a validated compiler if the vendor affirms that this compiler in fact conforms to the Ada standard.

A project validated compiler is a validated compiler which at some point in a project's life cycle is baselined for this project. Although the project validated compiler's validation certification may lapse prior to project completion, this compiler shall be considered validated for the duration of the project. A project validated compiler may be maintained or upgraded, as long as it continues to pass at least all of the ACVC tests applicable at the time its validation certificate was issued.

2.3 Activity Reports on Ada Programs

This section reviews ongoing Ada activities. It provides contact points where the information is available.

2.3.1 Programs Using or Designated to Use Ada

The following major programs, tentatively designated to use Ada in the Air Force Interim Policy on Computer Programming Languages, were removed from the list because they were found not to meet the requirements of Paragraph 2.a. (language commitment prior to 10 June 1983, or Milestone I or II reviews preceding 1 January 1984 or 1 July 1984 respectively):

- C-17
- HH-60D, Night Hawk

The following programs were granted an Ada waiver:

- Interservice/Agency Automated Message Processing Exchange (I-S/A AMPE)
- Sensor Fused Weapon (SFW)

The following program has requested a waiver, whose approval has been deferred until PDR:

- Joint Surveillance and Targeting Attack Radar System (JSTARS)

The following programs continue to be tentatively identified to use Ada:

- Wide-Area Anti-Armor Munition (WAAM)
- Enhanced Joint Tactical Information Distribution System (JTIDS)
- Joint Tactical Missile System (JTACMS)
- World Wide Military Command and Control System (WWMCCS) Information System (WIS)
- Advanced Tactical Fighter (ATF)
- Microwave Landing System (MLS), MIL-SPEC Avionics Segment
- Space Based Surveillance System (SBSS)
- COMBAT Identification Friend or Foe (IFF), MARK XV portion

2.3.2 Contact Points for Ada Information

Appendix 4 contains a list of contact points for Ada information. This section provides brief explanations of selected items from the Appendix.

The AJPO is a focal point for Ada policy. The Ada Information Clearinghouse provides information on MIL-STD-1815A, compilers, environments, validations, and education and training. The Ada Information Clearinghouse publishes a quarterly newsletter.

The Ada Integrated Environment, now renamed the Ada Compilation System, and the Ada Language System are Air Force and Army funded efforts, respectively, to produce Ada compilers and tools. More discussion of current DoD efforts in this area may be found in Section 6.2.

The Ada Validation Organization manages the Ada validation process and ensures that the validation policy and guidelines are consistently followed. The Ada Compiler Validation Capability is the test suite used in validating compilers and is kept under configuration and control at ASD.

DIANA is a machine-independent, intermediate language representation used by some compiler vendors. A fuller description of DIANA may be found in the Glossary of the Program Manager's Guide to Ada.

The CAIS is the Common APSE (Ada Programming Support Environment) Interface Set. It is briefly discussed in Section 3.4 of this Edition.

SECTION 3

RISKS AND RISK MANAGEMENT

Given the relative newness of Ada technology, there are risks associated with undertaking an Ada project. Because Ada is maturing fast, the risks have changed. This section examines these risks, in particular the impact of validation policy and the issues relating to run-time efficiency and support environments.

3.1 Impact of Validation Policy

There is an inherent tradeoff between the rigor of an Ada compiler validation policy and the timely availability of validated Ada compilers for a particular application. Because the resources available for performing validation are limited, more rigorous validation procedures increase the time between the completion of a compiler conforming to MIL-STD-1815A and its validation. Furthermore, stricter policies decrease the number of compiler versions considered validated and the number of target configurations for which a given compiler is considered validated.

Based on three years practical experience, the proposed new validation policies and procedures provide for more validated Ada compilers, available sooner. This decreases the risks associated with Ada compiler availability and are discussed further in sections 3.1.1 and 3.1.2 below. Section 3.1.3 discusses other risks affected by the proposed new policies and procedures.

3.1.1 Availability of Validated Compilers

The proposed policy would promote the availability of validated Ada compilers in several ways:

1. If a validated base compiler works without modification on a target configuration other than the base configuration, it need not be revalidated for the new target configuration. If simply registered as a derived compiler, the compiler will be considered validated. This expedites the production of validated compilers for an entire family of machines with closely related architectures.

2. If a vendor modifies a validated base compiler to make it work on a new target configuration, no revalidation is necessary. Again, the new compiler will be considered validated if it is registered as a derived compiler.

3. Maintenance of a validated compiler does not nullify its validation. Later versions of a validated compiler are considered validated, as long as the new versions are clearly distinguished and are not known to fail any ACVC tests.

4. A validated compiler (along with subsequent versions resulting from maintenance of that compiler) may retain its status as a project-validated compiler even after the compiler has lost its general validation. (A compiler can lose its general validation when the base compiler validation certificate expires, when a registered derived compiler is found to fail an ACVC test, or when the ACVC is modified in such a way that the compiler no longer passes all tests.) This permits a project to continue using the compiler selected at the start of the project.

5. A compiler for a restricted target machine may be considered project-validated even if the target machine is incapable of running all ACVC tests, as long as (a) the compiler is derived from a fully-conforming project-validated compiler for a generic target and (b) the derived compiler supports all mandatory language features that can be supported on the restricted target. This increases the availability of Ada compilers for embedded computers with limited I/O capabilities and memory.

6. A project-validated compiler may be derived by tailoring run-time libraries to the needs of a particular application, as proposed in Section 3.3 of this document. As long as the application-specific library is used only within that application, the compiler with the original run-time libraries remains project-validated for other applications. This increases the availability of project-validated compilers capable of meeting the time and space constraints of a particular application.

7. The term of a validation certificate, now fixed at one year, will be determined by the Director of the AJPO, who can extend this period if the workload on Ada Validation Facilities and Ada compiler maintainers warrants such a change. In particular, such an extension would be warranted if a reinterpretation of language rules were to cause the ACVC to undergo a

major change, requiring major modification of many existing compilers. These compilers would retain their status as validated compilers throughout the extended term, while modifications were implemented.

3.1.2 Reliability of Validated Compilers

Testing is never a foolproof way of ascertaining the correctness of software. It has always been the case that compilers containing errors could pass all ACVC tests. Furthermore, a compiler that conforms to MIL-STD-1815A is not necessarily usable in a practical sense. In the words of the proposed Validation Policies and Procedures,

> Users are responsible for understanding the scope and limitations of compiler validation, which is a means to increase confidence in the conformity of an Ada compiler to the Ada language standard. While such conformity is a first measure of usability of the compiler, it by no means guarantees that a Validated Compiler satisfies all the usability requirements of a particular project.

Just as the proposed Policies and Procedures make it easier for a compiler used on a particular MCCR project to be considered validated, so they increase the chance that a nonconforming compiler will be inadvertently considered validated. Designation of a compiler as project-validated increases confidence that an Ada compiler conforms to MIL-STD-1815A, but stricter policies would result in higher confidence.

The risk of error in a validated compiler has several sources:

1. Though the AJPO may require supporting information from a compiler vendor, registration of a derived compiler as validated rests principally on an affirmation by the compiler vendor that the derived compiler conforms to MIL-STD-1815A. The vendor has a vested interest in this determination, and there are no objective standards or procedures for making this determination.

2. There may be subtle differences between the base configuration and a closely-related configuration. A vendor unaware of these differences may affirm that a validated base compiler conforms to MIL-STD-1815A under both configurations, and no ACVC testing is needed to confirm this.

3. There are no guidelines requiring a derived compiler to be closely related to its base compiler. Errors may be introduced through major changes. Nonetheless, the derived compiler need not be subjected to ACVC tests.

4. Violations of MIL-STD-1815A may be introduced through maintenance of a validated compiler, but the maintained compiler is considered validated without further ACVC testing.

5. A generic target that is thought to be a superset of some restricted target may in fact not be, so that a compiler validated on the generic target may not behave correctly on the restricted target. ACVC testing may be impossible on the restricted target.

6. Replacement of the run-time library used to pass the ACVC with an application-specific run-time library may introduce errors but may also make ACVC testing impossible.

7. If the term of validation certificates is extended, compilers will have to pass updated ACVC tests less frequently.

The risk of an incorrect compiler passing the ACVC will decrease steadily over time, though it will never reach zero. As errors in validated compilers are discovered, the ACVC is augmented with tests that would have caught such errors. Since validation certificates periodically expire, compiler vendors must eliminate such errors if their compilers are to remain validated.

The greater risk is that, because compilers can be considered validated without undergoing full ACVC testing, compilers incapable of passing the ACVC tests may be used in projects. To minimize this risk, program managers should recognize that there are different levels of validation, implying different degrees of confidence in a compiler's conformance to the Ada standard. These levels can be ranked as follows, starting with those compilers providing the highest degree of confidence:

1. Validated base compilers, generating code for base configurations. (Such compilers have current validation certificates earned by passing a recent version of the ACVC on the base configuration.)

2. Registered derived compilers that are really unmodified base compilers generating code for configurations other than the base configuration

3. Registered derived compilers that are variations or revisions of base compilers

4. Project-validated compilers for a generic target

5. Variations and revisions of project-validated compilers, including compilers for restricted targets

The proposed Policies and Procedures recommend that the initial selection of a project-validated compiler be contingent on that compiler passing all ACVC tests. If the program manager follows this recommendation, a newly project-validated compiler will be as reliable as a validated base compiler. Nonetheless, project-validated compilers are not subject to periodic revalidation, so a generally validated compiler may over time become more reliable than a project-validated compiler.

3.1.3 Other Risks

At the time of this writing, the Validation Policies and Procedures are still in draft form, undergoing public review. Changes to the draft can be expected. In the very short term, uncertainty about the ultimate policy to be adopted poses a risk.

There are more serious risks as well:

1. Designation of a particular version of a particular compiler as project-validated does not require the compiler vendor to continue to support that compiler or to keep later versions compatible with the version chosen as a project baseline. The project may be forced to assume maintenance of the compiler or to adopt unnatural coding practices to avoid compiler errors.

2. The project must assume the burden of choosing a project-validated compiler, performing acceptance testing, and repeating the acceptance testing at each baseline milestone in the maintenance cycle if the current compiler version has not been subjected to appropriate testing.

3. The designation of a compiler as project-validated expires with a major system upgrade. The risks of selecting and testing a new project-validated compiler must be confronted at that time. The new project-validated compiler must be one that is validated at the time of the major system upgrade. It must reflect any language changes that have taken place since the last project-validated compiler was selected.

The proposed Policies and Procedures reduce the risk that a compiler vendor will avoid the repair of known errors for fear of having to revalidate the compiler. A maintained compiler retains its status as a validated compiler, subject to the vendor's affirmation that the compiler continues to conform to MIL-STD-1815A.

3.2 Run-time Efficiency

Fast execution of an Ada program depends on the quality both of the code generated by the compiler and the run-time system invoked by that code. The Ada language allows the generation of very efficient object code. The Ada compiler marketplace has become highly competitive, and evidence suggests that, month-by-month, the quality of the code actually generated by Ada compilers is quickly improving.

Nonetheless, the size, speed, and algorithms of the run-time system may be the key to meeting the time and space constraints of many Ada applications. In many cases, a run-time system may have to be customized to the needs of the particular application, an eventuality anticipated by the proposed Validation Policies and Procedures. This approach is feasible to the extent that a compiler's run-time system has individually replaceable modules with well-defined interfaces. The issue of custom-tailoring run-time support environments is dealt with in greater detail in Section 3.3.

Section 3.2.1 below addresses the potential for efficient code generation. Section 3.2.2 identifies the features that pose the highest risk for fast execution and provides guidelines for managing that risk.

3.2.1 Ada Language Support for Efficient Code Generation

The designers of the Ada language cite efficiency as one of their principal design goals. The language design reflects a strong awareness of current machine architectures. Language rules provide many opportunities for compile-time analysis and optimization. In addition, the Ada programmer has a degree of direct control over code generation, to further improve efficiency.

An awareness of the underlying machine is reflected in the design of the Ada language by features like the following:

1. Scope rules for loop indices that allow efficient code generation for a wide variety of instruction sets

2. Rules allowing a compiler to choose the most efficient representation for a numeric type, once the programmer has abstractly specified the minimum range and precision requirements

3. Rules facilitating the implementation of fixed-point real arithmetic using integer arithmetic in machine instructions

4. Description of private type representations in package specifications rather than in package bodies (which would logically be more appropriate), because description in package specifications allows the generation of more efficient object code for modules using the private type

5. Multitasking rules that deliberately allow a wide variety of implementations, compatible with different architectures and application requirements

Opportunities for compile-time analysis and optimization include the following:

1. Compile-time evaluation of expressions, compile-time analysis to eliminate run-time checks, and dead-code elimination are specifically mentioned in section 10.6 of the Ada Language Reference Manual.

2. Section 11.6 of the Ada Language Reference Manual has several rules ensuring that the possibility of exceptions being raised does not prevent optimizations. Classical optimizations are allowed even if they would cause a program to behave differently from an unoptimized program when exceptions are raised.

3. The detailed type and subtype information found in an Ada program can make certain optimizations easier to find, eliminating the need for many run-time checks and allowing certain Boolean expressions to be evaluated at compile-time rather than run-time. [Wel78] reported that analogous subrange information in Pascal programs allowed a fairly simple optimization algorithm to eliminate most run-time checks.

Furthermore, an Ada programmer may specify performance-improving beyond those provided by an optimizing compiler, including the following:

1. Selective suppression of run-time checks not eliminated by optimization.

2. Treatment of a subprogram call as a macro call, so that the code of the subprogram is expanded inline and the call does not incur linkage costs.

3. Specification of whether the primary optimization criterion for a particular program unit or a particular data type should be execution time or storage space.

4. Specification of a particular data representation efficiently implementable on the target hardware.

5. Control over how much storage is set aside for execution of certain tasks or dynamic allocation of variables to be pointed to by certain access types.

6. The ability to bypass Ada's normal type-checking mechanisms so that an Ada programmer can exploit bit representations in the same way as an assembly-language programmer.

7. The ability to control deallocation of dynamically allocated variables.

8. The ability to write small, critical portions of a program in assembly language or some other language.

3.2.2 Ada Features Posing Short-Term Efficiency Risks

In comparing the efficiency of Ada and earlier languages, one must recognize that the Ada language provides certain capabilities not provided by earlier languages. These capabilities are provided in such a way that, in a good implementation, they will incur no performance penalty if they are not used. It follows that programs in a language like FORTRAN can be transliterated into Ada with no significant loss of efficiency.

Some of the capabilities provided by the Ada language, but not by its predecessors, impose no additional run-time overhead. Such features include packages, programmer-defined types, strong type checking, private types, separate compilation without loss of consistency checking, overloading, and (arguably) exception handling. Such features do much to enhance the reliability and maintainability of an Ada program, but are processed almost entirely at compile-time.

Some features provided by the Ada language do potentially entail run-time overhead. These features are:

- Run-time checks
- Generic units
- Dynamic storage allocation
- Rendezvous

The efficiency of these features may vary widely from implementation to implementation. Within an implementation, the performance impact may vary from use to use.

Use of these features should not be discouraged, because they provide tangible software engineering benefits, and it is difficult to determine in advance whether or not a particular use will incur a significant performance penalty. Furthermore, programs that use these features can be transformed in a fairly straightforward way to programs that do not. Such transformations can be applied selectively, after a program has been constructed and the sources of inefficiency have been pinpointed by metering tools.

These considerations suggest the following strategy for minimizing efficiency risks:

1. A program should be written in an appropriate abstract Ada style, without concern for the efficiency with which certain features are implemented.

2. If the program fails to run fast enough, it should be metered to pinpoint the bottlenecks.

3. If a particular use of a particular feature is creating a bottleneck, that use should be eliminated by transformation. The original, untransformed program can then be viewed as a high-level design from which the final program was derived. The untransformed program should be retained as a design document.

4. Later program modifications should be applied to the untransformed program, and the transformations should then be repeated to the extent they are still applicable, to generate a new version of the program.

5. Later improvements in hardware or Ada compilers may make the higher-level, untransformed program usable directly for the generation of efficient code.

Project planning must account for the time that will be necessary to meter and transform programs.

The remainder of this section describes specific performance issues and transformation strategies.

3.2.2.1 Run-time Checks

In principle, the Ada language requires many run-time checks. As noted earlier, many of these are eliminated by a good optimizing compiler. Most of the remaining checks will not significantly impact performance. Those that do can be eliminated by a simple transformation -- the introduction of a Suppress pragma.

Correct handling of unanticipated exceptions may be crucial for an embedded application to terminate harmlessly in case of a software error. Therefore, Suppress pragmas should be applied to a limited region of the program, and this region should be carefully scrutinized to ensure that the checks being eliminated are indeed unnecessary. Empirical studies [Knu72] show that a program spends most of its time executing five percent of the program text. Improvements in this small region of the program will significantly improve performance of the overall program, but improvements elsewhere will have minimal impact. The role of metering is to pinpoint the parts of a program that are critical to overall performance.

3.2.2.2 Generic Units

There are many different ways to implement generic units [Bra83]:

- Making a separate copy of the generic template for each instantiation, so that each instance is as fast as a nongeneric program unit, but each instantiation greatly expands the size of the object code

- Compiling generalized object code directly from the template (for example, compiling an assignment as a loop copying a number of bytes specified in a control block set up during the generic instantiation), so that there is only one copy of the object code, but the code runs more slowly than code generated from an ordinary program unit

- A compromise in which different copies of the instance are made only for instances that manipulate different sizes of objects, so that instances dealing with data of the same size can share copies and each copy is as efficient as an ordinary program unit

An ideal implementation would provide the programmer with pragmas to control the instantiation mechanism on a case-by-case basis.

Generic instantiations in a design can be simulated manually in production code by editing a copy of the generic template and inserting it in place of the instantiation. The effect of sharing a copy applicable to many types can be achieved by unchecked conversion. This is a low-level approach that has many drawbacks, and it should be viewed solely as the pragmatic implementation of a higher-level construct.

3.2.2.3 Dynamic Storage Allocation

For some implementations, dynamic allocation may be a source of significant inefficiency, though there is no inherent reason why this must be so. Some allocators in Ada programs can be eliminated by the use of declarations inside block statements. In other cases, programmers can implement their own storage allocation schemes. This can be done either at the source level or by modification of the run-time system. Source-level implementation of storage management involves declaration of a large array to serve as a heap and the writing of allocation and deallocation routines tailored to the application.

3.2.2.4 Rendezvous

Rendezvous are intended to be the primary means of inter-task synchronization and communication in the Ada language, but current implementations of rendezvous are generally not fast enough for most real-time applications. Researchers have long been aware ([H&N80], [Hil82]) of the potential to implement certain patterns of rendezvous quite efficiently (often without need for a context switch) by exploiting pragmas in which the programmer notifies the compiler of the pattern. At least two validated Ada compilers now take this approach. Because of the competitive nature of the Ada compiler market, others can be expected to follow suit quickly.

In the long run, improvements in rendezvous speed may also come from hardware. One approach is an Ada-oriented machine with specific instruction-set support for rendezvous. Another [R&M76] is a multiprocessor architecture in which each application processor is accompanied by an "agent processor" that handles entry queue management and other housekeeping details for the tasks running on the application processor.

In the short run, rendezvous ought to be used as the first step in designing a multitask system. If the resulting performance is unacceptable, entry parameters can be replaced by shared variables. Entry calls will then serve solely to synchronize tasks. If performance is still unacceptable, entry calls can be replaced by application-specific synchronization primitives like semaphores. These can be invoked directly from the Ada language, by calling code procedures or application-specific run-time system routines. In a cyclic system, an Ada multitask design using rendezvous can be used as the underlying design before assigning specific work to specific cycles of a frame, as described in [Hoo86].

3.3 Alternatives in Custom Tailoring Run-time Support Environments (RSE)

The minimal Ada compilation system must include a Run-time Support Environment (RSE). The code generated by a typical Ada compiler calls on the RSE to obtain the run-time system services necessary for execution. The RSE consists of primitive and system data objects as well as a set of routines that provide functionality not supported by the target computer system. On a bare target the RSE plays the role of a virtual operating system. It provides task scheduling services to implement Ada language constructs like entry calls, accept statements, delay statements and selective waits.

Recent research on developing real-time systems in Ada has shown the need for a mechanism whereby a completed system can be tuned to meet specific real-time requirements. In evaluating the risks of an Ada project, a project manager must understand the role and functionality of the run-time system. Significant performance improvements in the Ada code can result from judicious use of tools provided in the run-time system. Three alternatives are presented in the following subsections: unique RSEs for each application, smart linking, and run-time customization.

3.3.1 Unique RSE for Each Application

Embedded computer system applications often have space and performance requirements that are not necessarily satisfied by the standard RSE configuration of an Ada compiler. One way to meet these requirements is to customize the RSE. Certain aspects of an Ada program's behavior, including scheduling and storage-allocation algorithms, are not completely determined by the rules of the language. It is therefore possible for a single compiler to have several alternative RSEs. A typical

example would be to substitute time sliced with strictly preemptive scheduling in the RSE (or vice versa, depending on the application). Other examples might include special memory allocation schemes (such as a multiple heap vs. a single heap scheme), or support for specialized entry call optimizations (such as fast interrupt entries).

3.3.2 Smart Linking

In some implementations the elaboration of the library packages of the RSE may generate references to practically all of the RSE packages, even though the application might only need a subset. A typical alteration would minimize the RSE to include only those functions required by the systems design constraints. For example, it is highly unlikely that an embedded application will require high level I/O or time management support. To meet memory constraints these functions might be omitted from the RSE. The obvious solution is to link to the application code only those RSE modules which are called. Deleting functions from the RSE, however, is not a simple task. It may require changes to the entire run-time environment because of interpackage dependencies within the RSE. There is ongoing research in software reconfigurability techniques, specifically in ways of removing portions of the RSE from the executable image.

At present, "smart linking" to the RSE modules required for a particular application is usually done through multiple statically configured RSEs. For each new set of application requirements for run-time support, another static configuration of the RSE is created. This is costly and difficult to maintain.

Current research in software reconfigurability is investigating alternate approaches. One solution is to remove the interpackage dependencies during the compilation of the RSE. Another approach is to build a flexible RSE so that the compiler user, instead of the vendor, tailors the RSE configuration. This effort will be facilitated through the work of the Ada Run-Time Environments Working Group (ARTEWG), a working group of SIGAda which is studying the standardization of RSE interfaces. At the time of this writing, ARTEWG is chaired by Mike Kamrad, whose address is:

Honeywell
M/S MN65-2100
3660 Marshall St. NE
Minneapolis, MN 55418
(612) 782-7321
Kamrad @ HI-MULTICS

3.3.3 Run-time Customization

Validation of a program's timing characteristics requires sophisticated timing analysis tools in the run-time support environment. These profiling tools should measure program operation, throughputs and response time. Additional tools will be needed in order to identify the cause of a timing problem, recording information such as the task interactions and scheduling decisions. This will allow specific alteration to be made in the application code and run-time system to solve any particular run-time problem.

The ability to modify the timing performance of a system is crucial to all successful real-time system development. Depending on the desired behavior, different solutions may be appropriate, ranging from increasing system throughput to trading speed for more deterministic timing characteristics. The RSE can provide extensive support for the range of tuning actions. Specifically, the following run-time customizations would be useful:

1. Modify the scheduler to eliminate pathological cases of inter-task interference.

2. Provide fast versions of some of the support programs, for use in special cases.

3. Support source level optimizations through run-time system capabilities (such as replacing monitor tasks with semaphore operations).

3.3.4 Costs of Custom Tailored Run-time Systems

Custom tailoring of run-time systems may be necessary in many cases but the cost is always high. Three of the most important impact areas of this cost are:

- Development cost
- Portability/Reusability
- Verifiability/Reliability.

Development costs are high because a run-time system must be worked on as well as the application code.

Portability and reusability are affected because the software is relying on custom enhancements to meet performance constraints. When the software is moved to a new environment (either to be reused on another project or to be ported to a new system) there is no guarantee that it will work because it may not have the same custom enhancements.

Verifiability and reliability are threatened by multiple implementations of the same run-time feature (such as the scheduler) that differ in subtle ways. It becomes difficult to build a precise statement describing program behavior, making the program difficult to verify. This in turn leads to possible reliability problems with software that has not been adequately validated.

3.4 Tool Portability and the CAIS

The Common APSE Interface Set (CAIS) is intended to promote portability by defining a standardized interface between tools and the Kernel Ada Programming Support Environments (KAPSEs). A KAPSE defines a virtual operating system, including such services as file management, input/output, communications, and process control. Section 2.2.1.3 of the Program Office Guide to Ada explains the purpose and concept of a KAPSE more fully.

The KAPSE Interface Team (KIT) and KAPSE Interface Team from Industry and Academia (KITIA), collectively composed of representatives from government, industry, academia, NASA, and foreign governments and institutions, has produced the proposed CAIS standard, issued in January 1985. This standard is undergoing public review and is expected to become a MIL-STD.

The goal of the CAIS is to promote source-level portability of Ada programs across DoD APSEs. The CAIS document defines two key concepts ([KITPR]):

> Interoperability is defined as the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion. Transportability of an APSE tool is defined as the ability of the tool to be installed on a different KAPSE; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming.

These interrelated goals are extremely important in increasing the cost effectiveness of the software development process.

The work of the CAIS has entered a second phase, whose goal is to continue to refine the existing CAIS standard as well as to resolve issues deferred by the original document. The AJPO, through the Naval Ocean Systems Center (NOSC), has awarded SofTech, Inc. a contract to continue the CAIS

development. A draft and final CAIS 2 Standard will be produced, along with a Rationale, a Guide for CAIS Implementors, a Formal Semantic Description, and a prototype. The tentative publication schedule is:

Early 1987:	Draft CAIS 2 and Rationale
Early 1988:	Final CAIS 2 and Rationale
1989:	Implementor's Guide, Semantic Description, prototype

Both the draft and final standards will be developed with responsive public review.

Issues addressed in the CAIS 1 effort focused on specifying the interfaces for the major structural elements, in particular for the data structuring model, the process control model and input/output. Major issues to be addressed in CAIS 2 include multilevel security, distributed environments, a sophisticated file system, a database typing mechanism, robust access control, history of database objects, a standard data interchange format, and interprocess synchronization and communication. CAIS 2 does not require a specific configuration management capability; however, the structure of CAIS 2 should allow several methods to be supported.

The CAIS is intended to be a long term, portable environment. This project is still in the requirements stage, thus its design is not yet known. Although there exist ongoing efforts within industry to develop early CAIS prototypes, a fully viable CAIS will not become available until the mid 1990's. Consequently, mandating the use of the CAIS on an Ada project during the next few years will entail a high risk for the project.

The second Edition will address the CAIS effort and portability issues in more detail.

SECTION 4

TRAINING AND RETRAINING

In order to achieve the transition to software engineering with Ada, program managers must address the issue of training, or in some cases retraining, their personnel. Section 4.1 explains the need for balanced training addressing not only the Ada language but also software engineering methods and programming support environments. Section 4.2 compares and contrasts the three primary training media, namely live instruction, videotape, and computer aided instruction. The extent of commitment, in terms of knowledge level and time, is analyzed in Section 4.3. Section 4.4 raises issues about measuring the effectiveness of the training. The last two sections explore existing programs focused on Ada training.

4.1 Syllabus

The design of the Ada language is based on modern software engineering principles, including structured programming, information hiding, data abstraction, the distinction between a module's interface and implementation, strong cohesion within a module, weak coupling among modules, and the reuse of modules. The benefits of using the Ada language come not from using a new syntax, but from the fact that the language facilitates the application of these software engineering principles. Training in the use of the Ada language will be incomplete if it emphasizes syntax and does not address the underlying software engineering principles.

Besides addressing language rules and software engineering principles, an effective Ada training program must provide programmers and their managers with the practical tools they will need to construct Ada programs. This requires effective training in a software engineering methodology and an Ada Programming Support Environment.

A software engineering methodology is a set of concrete steps that can be followed to implement software engineering principles. Some methodologies are applicable during specific phases of the software life cycle while others span the entire life cycle. Some methodologies may be oriented towards specific application areas. Since the advent of the Ada language, several methodologies have been promoted as particularly effective when used in conjunction with the Ada language. These include Object-Oriented Design [Boo83], Process Abstraction Method for Embedded Large Applications

(PAMELA) [F&C86], and the use of an Ada-based Program Design Language [IEEE86]. Any of these methodologies would be appropriate for inclusion in an Ada training program.

Current Ada Programming Support Environments range from the bare essentials to sophisticated tools. At the least, an environment must provide a compiler, linker, loader, library manager, and run-time support. To meet the Stoneman requirements, moreover, an APSE must provide a set of tools which help automate all aspects of the software life cycle, including configuration management, documentation, project control, verification, debuggers, database manager, etc. In order to be productive, it is important that both programmers and managers understand the type and operation of the tools available to them. Furthermore, because APSEs may be extended through the addition of user-defined tools, environment training should address the writing and incorporation of new tools.

4.2 Media

As the use of Ada has become more widespread, the variety of training options has increased. Ada training is offered in lecture format, through videotapes, with computer-aided instruction (CAI), and in textbooks. The remainder of this section considers the first three of these media. Textbooks in and of themselves are usually not sufficient, but they are a necessary and vital supplement to all the other methods. ACM Ada Letters regularly publishes reviews of new textbooks.

Live instruction is used both in technical courses and in management seminars. Some technical courses provide additional lab time in which the students gain experience implementing Ada programs. Individual courses range in length from a half day to six weeks intensive study. Live instruction is the most effective training medium, in part because Ada training requires teaching knowledge (software engineering concepts) as well as skills (syntax and semantics). Students benefit from the interaction with the teacher and the ability to ask any kind of questions. Moreover, the instructors can assess those areas where individuals have conceptual difficulty, and they can dynamically adapt the material to address those needs.

Videotapes can be an extremely effective training medium. They can provide a good overview of software engineering principles and the capabilities of the Ada language that put these concepts into practice. Because they lack a hands-on component and student-teacher interaction, however, they do not fully answer the need for in-depth technical training. Furthermore, they may not

convince those programmers experienced in older languages such as FORTRAN, Assembler or COBOL who are resistant to Ada and to a new approach.

Computer-aided instruction has the advantage of being hands-on. It is most effective for technical training, giving the user practice in exercising APSE tools and writing Ada code segments. Unlike videotapes, there is some student-teacher dialogue, albeit one limited to the teaching program pointing out an incorrect response. The more sophisticated CAI packages try to explain not only what is wrong but also why it is wrong. CAI learning is essentially learning by example and by repetition. One of the problems is that the program cannot alter dynamically the explanation of some concept which a student finds difficult, as measured by how often each lesson is repeated. Furthermore, most CAI programs do not have the flexibility to allow students to create their own lesson plans; they must follow a predefined sequence.

Live instruction is the most intensive of the three methods. Courses are usually taught on consecutive days, so there is a risk of overloading the students. This method requires the commitment of larger blocks of the students' time, thus having an inevitable impact on project schedules. Videotapes and CAI, on the other hand, are self-paced. Because the time commitment is staggered, there is less adverse impact on project schedule. In general, some time should be anticipated for training and built into the project schedule.

Of the three media, live instruction is the most expensive. Whereas video tapes and CAI packages have a one time acquisition cost and offer the opportunity for infinite reuse, thus reaching the largest audience, live instruction is limited to a particular class, so the acquisition cost is incurred with each presentation.

4.3 Training Commitment

There are two aspects to training: who needs to know how much, and how long will it take. These questions are discussed in the sections below.

4.3.1 Audience

In preparing for the transition to Ada, personnel in many different job categories will need some level of Ada training. At a top level one can distinguish between managerial and technical training. Management awareness and commitment at all levels is crucial to the success of an Ada project. Executive management training should focus on the

transition issues, and on the benefits and costs of using Ada. A very brief language overview is appropriate. At lower management levels, the training emphasis should be on successfully running an Ada project. These personnel should be familiar with the purpose of Ada's major features as well as with the applicable software development methodologies. They should also understand the more technical aspects of transition issues, such as environment selection, technical training needs, methodology selection, compiler validation issues, portability, and reusability.

Technical training needs range from the introductory through the advanced. Junior personnel can become productive after some initial hands-on training, using the same environment they will use on the project. They do not need to know the whole language in order to code from the detailed design. It is important that this training stress good software engineering practice so that they acquire good habits and the "Ada mindset" from the beginning. Senior personnel and system designers should receive thorough training in both language and methodology. They should understand the design tradeoffs of Ada's program structuring features. Quality assurance and testing personnel also need advanced Ada training. QA should understand the characteristics of a good design and testing should be familiar with program verification techniques.

In addition to the management and technical personnel training needs, other support personnel, such as contracts and configuration management, should also receive some formal exposure to the Ada solution. Contracts personnel should understand the motivation for using Ada, the need for a "total" approach encompassing software engineering, tools and language, as well as the cost impact in different phases of the software life cycle.

Configuration management personnel should understand the program structure facilities of the language. They should also understand the compilation ordering rules and the impact of program changes on recompilation requirements. Moreover they should be familiar with the configuration management, documentation, library and database tools of the APSE.

Current experience has shown that recent computer science graduates (1980's) are very well prepared to learn Ada and modern software engineering techniques. A small but growing proportion are able to take an Ada course as part of their curriculum. Among experienced programmers, on the other hand, Ada project managers may find a combination of resistance to change and difficulty in mastering new concepts. Veteran programmers will have to unlearn their familiar program and data structuring techniques so that they do not write, say, FORTRAN code in Ada syntax.

4.3.2 Time Requirements

In-depth technical training can seldom be accomplished in the space of a week - there is too much material to cover in more than cursory fashion. Because the most effective training combines both lecture and lab time, Ada project managers should expect that their most senior people will spend several weeks undergoing Ada software engineering training. In order to achieve a high retention rate, the training should not occur over consecutive weeks. Ideally it would be scheduled so that students have the opportunity to practice on the job what they have learned in class.

4.4 Evaluation Criteria

Current training offerings vary widely with regard to their student evaluation mechanism. Some courses award certificates based on attendance while other programs administer formal testing. These tests stress a student's mastery of Ada syntax and semantics.

At present there is much interest in a formal mechanism through which to ascertain Ada competency. Analogies have been drawn to other disciplines which require formal certification. The key issues in the certification debate are:

- Should it be done?
- Who should administer certification (AJPO, Institute for the Certification of Computer Professionals)?
- Are different levels of certification needed?
- What are the costs associated with certification?

Three groups are independently investigating the topic of certification: the Education Subcommittee of SIGAda, the Ada Software Engineering Education and Training (ASEET) task force, and the Armed Forces Communications and Electronics (AFCEA) Ada Education and Training Study (ADETS) study. Section 4.6 below describes the work of ASEET and ADETS in greater detail.

4.5 Existing DoD and Air Force Programs

Within the Air Force there are three centers providing Ada education and training, namely Keesler AFB, Air Force Institute of Technology (AFIT) and the Air Force Academy. A

fourth curriculum, developed under Army sponsorship, is available through the Defense Technical Information Center (DTIC).

The Air Training Command has developed 4 courses at Keesler AFB. They are:

Ada for Executives	- 1 Day
Ada Managers Orientation	- 4 Days
Ada Project Manager	- 10 Days
Ada Application Programmer	- 6 Weeks

The first three courses may be taught at Keesler, or on site using the Mobile Training Team. The programmer's course is only offered at Keesler. Future courses planned will address low level implementation and system design. Over 700 people were trained in Fiscal Year 1985. Keesler has a capacity to train up to 3000 people per year through the expansion of their instructor staff. The point of contact is Mary Rivers at AV 868-3110 or (601) 377-3110.

AFIT is offering two graduate courses in their Ada curriculum. There is an introductory computer science course which gives students an overview of the Ada language, emphasizing the program structure built through packages. The second course, "Effective Programming with Ada," discusses the entire language, with emphasis on data structures. An advanced course is planned, focusing on the APSE and on developing special projects. AFIT trained 125 in Fiscal Year 1985 and has a capacity to train 150. The point of contact is Lt. Col. Rick Gross at AV 785-3098 or (513) 255-3098.

The Air Force Academy has a computer literacy requirement; however, it currently teaches Pascal in this core course. The Academy continues to offer Ada seminars to interested students. The point of contact is Maj. James Nielson at AV 259-4112 or (303) 472-3590.

The Army has investigated Ada curriculum requirements, leading to the development of 9 language modules and 4 software engineering modules. Supplementary materials include 3 workbooks containing tutorial, exercises, and annotated solutions as well as a set of case studies. The Defense Technical Information Center (DTIC) Accession Numbers for these course materials are listed in the table below. The numbers in parentheses indicate the length of the course:

Ada Primer		A165345
Advanced Ada Workbook		A146257
Real-Time Ada Workbook		A146258
Ada Case Studies II		A140818
Ada Orientation for Managers (L101)	(1)	A165351

Ada Technical Overview (L102)	(1)	A165352
Introduction to Ada - a Higher Order Language (L103)	(1)	A141848
Ada for Software Managers (L201)	(3)	
Volume I		A165314
Volume II		A165315
Basic Ada Programming (L202)	(5; 10 with lab)	
Volume I		A166366
Volume II		A166367
Lab Manual and Exercises		A166043
Advanced Ada Topics (L305)	(5; 10 with lab)	
Volume I		A165075
Volume II		A165076
Volume III		A165077
Exercises		A165288
Real-Time Systems in Ada (L401)	(5)	
Volume I		A166351
Volume II		A166352
Using the Ada Language Reference Manual (L402)	(2)	A143582
Software Engineering for Managers (M101)	(1)	A165123
Exercises (M101)		A145094
Introduction to Software Engineering (M102)	(2)	A165122
Exercises (M102)		A144237
Software Engineering Methodologies (M201)	(5)	
Volume I		A165300
Volume II		A165301
Volume III		A165302
Workbook		A165299
Programming Methodology (M203)	(2)	A143581

4.6 Catalog of Resources for Education in Ada and Software Engineering (CREASE)

The Catalog of Resources for Education in Ada and Software Engineering (CREASE) is a comprehensive listing of Ada training materials offered within industry and government. It is published annually by the AJPO, and Version 4.0, May 1986, is the current version.

The CREASE organizes the training by the following categories: CAI, Lecture/Seminar, Informational Resources, videotape, and textbooks. The course listings are further subdivided by class of offering: company, government agency, and university. The amount of information given in each listing is determined by the offeror and does not represent the opinions of the AJPO or DoD. The information typically includes:

- o [course] objective
- o concepts covered
- o intended audience
- o prerequisite
- o course materials
- o offeror name, address, and background
- o price

4.7 Ada Education and Training Studies

Three organizations are chartered with investigating Ada education and training: Ada Software Engineering Education and Training (ASEET), Armed Forces Communications and Electronics Association (AFCEA) Ada Education and Training Study (ADETS), and the Software Engineering Institute (SEI). Both ASEET and ADETS are tasked with analyzing needs and making recommendations. The ASEET Team is a tri-service group chaired by Maj. Samuels of Keesler AFB at AV 868-3728 or (601) 377-3728. It was established in 1985 to identify a comprehensive training plan for DoD. This plan will begin with a needs analysis to determine learning objectives in addition to numbers of personnel needing training at what level. ASEET will study ongoing research projects, coordinate course materials, and examine certification.

The ADETS group was formed in late 1985 and is chaired by Frank Druding of Ford Aerospace, Palo Alto, California. Its purpose is to provide information and recommendations to assist DoD and industry in providing the education and training required for introducing and supporting the Ada language. Topics to be addressed include training requirements, capacity, shortfall, and certification. As part of the data collection task, the ADETS team will examine a cross-section of existing major Ada programs from both DoD and contractor perspectives. The ADETS findings will be published in a report in December 1986.

The SEI was founded in the early 1980s to accelerate the transfer of Ada technology. It is located at Carnegie-Mellon University in Pittsburgh. Its overall mission is to establish a standard of excellence for the art and practice of software engineering. One of its tasks is to create a Master's Degree program in Software Engineering.

SECTION 5

ADA PROGRAM DESIGN LANGUAGE

A Programming Design Language (PDL) is a formal notation used for describing software design. A PDL increases productivity, improves software quality, and minimizes the risks in development and maintenance. It is important for a PDL to be abstract and flexible, as well as to be able to express naturally modern design methodologies. A PDL must encourage a high level of abstraction; otherwise it could lure designers into specifying too much detail too soon, in which case they would be programming instead of designing. A PDL must be flexible; it should not place artificial constraints on the design, thereby discouraging designers from considering designs that could not be conveniently expressed in the PDL.

Section 5.1 discusses three issues relating to Ada PDLs: the use of automated tools, the advantages and disadvantages of compilability, and management benefits. Section 5.2 reviews the IEEE PDL working group activities as well as several DoD inspired PDLs.

5.1 PDL Issues

Ada is abstract and flexible. It contains features that support modularity, abstraction, information hiding, concurrent processing, reusability, and error processing. These features, which are lacking in the commonly used HOLs prior to Ada, reflect the goals of modern design methodologies and make Ada an ideal base for design language. An Ada-based PDL could be a subset of the Ada language, or it might contain language extensions. The benefits of an Ada-based PDL, which derive from the fact that the attributes of a good PDL are incorporated into the Ada language, can be enjoyed regardless as to whether the intended implementation language is Ada or another HOL.

If the implementation language is not Ada, a correct translation of a consistent design will produce a consistent program. Ada constructs that are not easily translated into the implementation language should be identified and avoided. When the implementation language is Ada, there are additional benefits such as

- An easier transition from design to code
- The ability to apply a common set of tools and methods to both the design and the program
- Reduced costs of training.

5.1.1 Automated PDL Processing Tools

Automated tools to process the design produced by an Ada-based PDL are crucial if the benefits of increased productivity and design correctness are to be achieved. The advantage of analyzing the design automatically is to provide an early check on the consistency of the software interfaces. The validation of design consistency before coding and the performance of semantic analysis, such as data flow analysis, will minimize the risks through the entire software life cycle.

There are two ways of fulfilling the need for automated PDL processing tools. If the PDL contains no extensions to the Ada language, such analysis can be performed by an Ada compiler; otherwise a special PDL processing tool must be built. The compilability of an Ada-based PDL is a controversial issue. The benefits of a compilable PDL are designs that can be processed by standard Ada tools, and a more straightforward mapping from designs into Ada programs. Furthermore, the compiler output could be treated as a prototype of an executable design, revealing design errors at an early stage.

A compilable PDL, however, can have an adverse effect on the design process, whereas special-purpose PDL tools do not. Compilable PDLs encourage premature coding and are ill-suited for expressing evolving designs. An Ada compiler cannot check for structured comments which convey design information. The primary goal of a compiler is to generate efficient object code, a goal which is not relevant in the case of PDL. If a compiler is used to process PDL, then the PDL must be complete and all entities fully declared. At the design level, however, it is neither necessary nor advisable to give the full implementation of a private type. Yet, the designer using a compiler as a tool must provide some skeleton declaration in the private part of the package, adding a level of detail to satisfy the compiler rather than to elucidate the design. Finally a special PDL processor, unlike a compiler, could combine Ada constructs with more abstract design notation, such as an iterator that processes each element of a collection.

Current thinking, as evidenced by the guidelines set forth in the IEEE Recommended Practice (see Section 5.2), favors a compilable PDL.

5.1.2 Management Benefits

An Ada-based PDL can be used as a management tool; it provides information that helps the manager organize, coordinate and control large software projects. The

information most relevant to management during the development life cycle relates to project organization, planning, status tracking and design review. This information, in a document or report form, can be obtained from analysis of the design description by various design analysis tools. Some of the reports produced by these tools relevant to management include completeness, consistency, design reliability, design review, performance prediction, project history, and software metrics. A more detailed description of these tools is provided in the IEEE Recommended Practice. Traditionally, most of this information was gathered during the implementation phase; however, an Ada-based PDL makes this information available earlier, during the design stage.

An Ada-based PDL subdivides the overall system design into separate units. Partitioning the design in this way is very helpful to the project manager because it supports the division of work among team members. The explicit dependencies between the design units make it possible to derive work scheduling information from the design information. Completeness criteria for design units are established in the design methodology. A unit might be considered complete, for example, if it satisfies requirements such as having interfaces named and logically typed, having functional descriptions, and having complete specifications. These criteria may be useful for status reporting and tracking.

An organization that uses an Ada-based PDL has made a visible commitment to the DoD directives for producing high quality, maintainable software. Inevitably, such an organization is adjusting its software design and development practices when it chooses to capture a design in an Ada-based PDL. At the same time, the organization is acquiring expertise in Ada and modern software engineering principles and in the case of contractors, improving their competitive edge and viability. The long term benefits of this transition are increased productivity and improved software quality.

5.2 Studies on Ada-based PDLs

There are numerous studies on the use of Ada as a PDL. The IEEE working group on "Ada as a Program Design Language" has defined guidelines for evaluating and developing a PDL based on the syntax and semantics of Ada. According to these guidelines, an Ada-based PDL should be compilable. Any extensions to the Ada language should be in the form of commentary text, flagged by a special indicator. Besides describing design language characteristics, features, and support tools, this study also covers management issues. Reballoting on this document is taking place in June 1986,

because only 75% of the ballots were returned in Oct 1985. Meanwhile the draft has been updated to accommodate the responses from the first ballot. The new ballots have been sent only to those who responded to the first ballot, and because the majority had voted affirmatively, passage is likely.

The World Wide Military Command and Control System (WWMCCS) Information System (WIS) program has developed guidelines for the Ada Design Language (ADL), which is a PDL that uses fully compilable Ada. ADL is a single design notation that will be used by a wide variety of organizations designing WIS systems. These organizations will be able to tailor the ADL to some extent in order to meet a particular design goal. Guidelines for such tailoring, which must be done within Ada itself, are provided. Structured comments that provide design information not readily expressed in Ada will be used. Annotations are to be distinguished from ordinary comments by a "sentinel character," and from each other by predefined keywords.

The Joint Interoperability Tactical Command and Control System (JINTACCS) Automated Message Processing System (JAMPS) Program Design Language (JPDL) is another Ada based PDL, which is used in the Air Force's effort to reimplement JAMPS in Ada. JPDL contains extensions to Ada such as predefined library units and additional pragmas. These are ignored by an Ada compiler, which makes JPDL compilable. Mappings from JPDL to FORTRAN and C are described because, even though JAMPS itself will be written in Ada, implementations using FORTRAN and C will use the JAMPS database.

An Ada-based PDL survey was performed for the Naval Avionics Center (NAC). Twenty-five separate companies involved in Ada-related work were contacted, and guidelines were developed for the Navy's development of an Ada PDL.

The Ada Design Language Developers Matrix is a useful source of information on the availability and the ongoing development of Ada-based PDLs by various organizations. The matrix is updated periodically in the ACM Ada Letters. It provides information on the closeness of the PDL to Ada and on the scope of the tool development effort. The next complete matrix will be published in the November-December 1986 Ada Letters.

SECTION 6

ADA COMPILER AND ENVIRONMENT TECHNOLOGY STATUS

As increasing numbers of validated compilers become available, the focus among Ada users is shifting from merely validated compilers to production quality validated compilers. This section points to sources of information on validated compilers and highlights the ongoing performance evaluation work.

6.1 Validated Host/Target Combinations

The number of validated host/target combinations is increasing rapidly, and any list included in this report would shortly become obsolete. Up-to-date information may be obtained either through the Language Control Facility Ada-JOVIAL Newsletter or through the Ada Information Clearinghouse. Points of contact for both organizations may be found in Appendix 4.

6.2 Work in progress

This section updates the status of Ada compilers under development by the U.S. Government.

	Host Computer and Operating System	Target Computer and Operating System
Air Force: (AIE, renamed Ada Compilation System)		
RADC AFWAL	IBM 370 family VAX/VMS	IBM 370 family MIL-STD-1750A
Air Force: (other)		
AFATL	Cyber 176 NOS or NOS/BE	Zilog 8002
ESD	VAX/VMS (ALS retarget)	INTEL 8086
Army: (ALS)		
CECOM	VAX/VMS	VAX/VMS
Army: (NYU Ada/Ed Interpreter)		
CECOM	VAX/VMS	VAX/VMS

Navy: (ALS/N)

NAVSEA	VAX/VMS	AN/UYK 44 and 43

The Air Force compilers listed above have all been validated with the exception of the 1750A target. Both Army compilers have been validated. The Navy compiler is still under development.

6.3 Performance Evaluation

The three most important measures for evaluating compiler performance are compilation speed, execution speed and run-time memory requirements. Compilation speed refers to the elapsed time required to turn source code into executable object code measured in lines per minute. Execution speed is the elapsed time required to execute a compiled program. Run-time memory requirements include run-time library, required I/O packages, and application data storage. The remainder of Section 6.3 presents guidelines in evaluating compilation and execution speed.

6.3.1 Compilation Speed

In comparing the compilation speed of Ada with another HOL, it is important to remember that the Ada compiler and run-time system are freeing the programmer from doing such tasks as ensuring consistency of package and unit specifications, instantiations of generics, resolution of overloading, and constraint checking for initialization. This in itself significantly decreases the programmer's burden, but the tradeoff is increased compilation time.

In comparing the compilation speeds of several Ada compilers, the following issues should be considered. The ease with which a compiler can be rehosted or retargeted varies inversely with compilation speed. Furthermore a compiler hosted on a Stoneman compliant APSE must meet more stringent data collection requirements (i.e. statistics, history, and other information on the program submitted to the compiler), exacting a penalty in compilation speed.

6.3.2 Execution Speed

Ada performs extensive run time checks. In comparing execution speeds between Ada and another language, it is important to qualify what Ada run-time checks are performed during benchmark execution. Because other languages provide very limited run-time checks, if any, the Ada test results could be distorted. When general execution speed is the primary evaluation criterion, then comparable checks should

be inserted into the benchmark source code in the other language. If real-time performance is the primary evaluation criterion, then all run-time checks should be suppressed. Run-time checks, however, should never be suppressed during software development and testing. Suppression of run-time checks should only be used as a measure of last resort in performance tuning. Global and local optimization provide a superior alternative for improving the run-time performance.

There should be a good language correspondence in the benchmark on the algorithmic level. Certain Ada language constructs may be impossible to translate because they do not exist in the other languages under consideration. For example FORTRAN lacks records and pointers, in which case the benchmark results may be misleading.

The frequency distributions of language constructs will vary among applications. At the beginning of a compiler evaluation it is important to isolate the language constructs used most frequently by the application and to find appropriate benchmarks which reflect them, because the effectiveness of a compiler for a particular application is determined by these constructs' demand for computer resources. Different combinations of features can have different performance characteristics. It is highly recommended both to perform more than one compiler benchmark test and to consider future application developments, as each application is a unique mixture of Ada constructs.

In most cases, benchmark results indicate both compiler and computer system performance. The compiler workload produced by a synthetic benchmark could have a bottleneck effect on the entire system configuration. This bottleneck does not necessarily reflect the overall resource requirements imposed by a particular application.

The Ada Evaluation and Validation (E&V) team at the Institute for Defense Analyses produced the Prototype Ada Compiler Evaluation Capability (ACEC), an organized suite of compiler performance tests as well as support software for executing these tests and collecting performance analysis data. The ACEC and other existing benchmarks will be covered in the second Edition.

6.3.3 Run-time Memory Requirements

Compiler technology is mature enough to handle the Ada data structuring capabilities in an efficient manner. The source for inefficiencies in memory utilization result from the size of the run-time library which is loaded prior to execution. Section 3.3 discussed run-time memory options in detail.

SECTION 7

CONVERTING NON-ADA PROGRAMS INTO ADA

The transition to Ada raises two problems: finding a way of reusing existing software written in another language, and cutting down the costs of redesigning and reimplementing existing software in Ada. There are no easy solutions to these problems. The desirable solution in some cases could be to create an interface between the existing code and Ada, as discussed in Section 7.1. Another solution, discussed in Section 7.2, is to attempt to convert the original code into Ada. This is a complicated process and in most cases, it will not produce the software engineering benefits associated with Ada.

7.1 INTERFACE Pragma

The life span of existing software written in another language could be expanded by interfacing the original with Ada. The interfacing can be done by using Ada's predefined INTERFACE pragma. The interface routine would link the non-Ada with the Ada run time environment, and it would allow non-Ada code to be invoked from an Ada program.

An implementation is not required to provide the INTERFACE pragma, and the implementations offered by different compilers will have different capabilities. Certain restrictions could be placed on this pragma, depending on the particular implementation of the Ada compiler. One implementation may allow full interface, another may allow partial interface, while others will not allow any interface at all.

7.2 Automated Translation

Transition to Ada may require conversion of some existing software to Ada. Short term costs are minimized by performing direct translation by hand or by partially or fully automated converters. These methods, however, are deceiving, because in many cases conversion entails long term costs, such as increased code size, memory space, and execution time, but most of all, reduced maintainability.

7.2.1 Expert Systems Technology

Some vendors are applying expert systems technology to the translation problem. Interactive transformers and source-code analyzers for this purpose now exist. The

resulting Ada source code, however, may be slower, less readable, and less maintainable compared to the original, especially for large complicated software systems that require a large amount of maintenance.

Conversion is not a simple line-for-line translation. On the contrary, it requires careful preparation and understanding of the code to be converted. The first step is to examine the architecture of the code that needs to be translated. The system design documentation is likely to be obsolete. The original design may have been modified in a way that reduces its coherence. The design may be impossible to translate efficiently into Ada without major redesign effort, in which case the translation project should be abandoned.

If it is decided to continue with the conversion then the conversion rules must be specified in detail. Conversion rules fall in the specific language constructs, and global language organization areas. Specifying rules for specific language constructs is often straightforward, but specifying rules for mapping overall program structure into Ada is more difficult. Programs with tasks, for example in RTL/2, are likely to be difficult to translate into Ada programs with tasks because the two languages are based on different models of concurrency and intertask communication. In addition, implementation strategies for the two languages may be different. Writing a conversion specification has a twofold benefit: it isolates problem areas and it is a valuable training exercise in both Ada and in sound software engineering principles.

A translation resulting in a FORTRAN or COBOL program written in Ada syntax should be avoided. Unfortunately, present day technology is incapable of converting an outdated design to an Ada-based design that concurs with current program design methodologies. The reason is that it does not exploit the advanced capabilities of the Ada language and does not provide the software engineering benefits that can be expected from the appropriate use of the Ada language.

7.2.2 Conversion of Artificial Intelligence Algorithms to Ada

The use of Ada in general artificial intelligence (AI) research is controversial. As with other languages, it is theoretically possible to convert any application from an AI language such as LISP, Prolog, Simula, or SAIL into Ada. In reality, the Ada code would be obscure, unreadable, and unmaintainable, and it would likely have poor run-time performance. The inefficiencies in the resulting Ada code stem from the lack of features in the Ada language necessary

to the efficient expression of AI techniques supported by the AI languages. These features include dynamically definable functions, subprograms as parameters, and garbage collection.

Dynamically definable routines are an inherent feature of AI languages, allowing function or program segments to be developed and executed at run time. This feature is used in creating programs that "learn." Ada lacks this capability. It is possible to simulate dynamic construction and execution of Ada code by implementing a table-driven machine; however, this approach is inefficient and less maintainable.

Many AI applications rely on using procedures as storable, denotable objects. Object templates can use a knowledge representation which includes names of procedures (or functions) that are called in order to instantiate the values of some of a particular object's characteristics. In Ada, it is not possible to store or pass procedures as objects. They are represented as control structures, rather than values of some data type. Attempting a direct mapping of this AI technique into Ada would sacrifice both efficiency and maintainability. A partial solution would entail building a specialized interpreter in Ada, which could understand the primitives from which a program is constructed.

Whereas AI applications routinely perform garbage collection, Ada does not guarantee it. For real-time applications, a sophisticated storage management system should be developed. Ada's private typing mechanism provides the primitives with which to build such a system.

Ada and AI are not incompatible. Artificial intelligence techniques and languages should be used to develop rapid prototypes. They promote unconstrained experimentation because they do not commit the application to the restrictions of an early design. Once both the problem and its solution are well understood, then the prototypes should be abandoned and the system should be built using a disciplined software engineering approach. The algorithms developed for the prototypes are transportable, although the actual AI code is not.

Ada offers AI applications several strong advantages. Strong typing promotes reliability. An Ada system will likely be more efficient than its AI counterpart because of the extensive Ada compile-time analysis. An Ada compiler has much more information at its disposal than, say, a LISP compiler, and the Ada compiler can therefore generate more

efficient code. Last and equally important, an Ada implementation offers coherence and understandability because of the control and data abstraction and of the modularity that are an inherent part of the language.

SUMMARY

The first edition of the Program Office Guide to Ada has focused on several aspects of risks and risk management. Key topics discussed include policy, with an in depth analysis of validation policy, and run-time issues, with special consideration of efficiency. The Guide has also addressed training options, the benefits of an Ada-based Program Design Language, and the implications of converting non-Ada programs.

Ada policy is being continually reviewed and updated to reflect the changing status of Ada technology. Waivers are becoming much more difficult to obtain. With the increasing number of validated compilers available for a greater selection of target processors, there is decreasing risk associated with procuring an Ada compiler.

The AJPO is coordinating a draft document on validation policies and procedures. These policies address the use of a validated compiler during a system life cycle. A compiler may be baselined and used for the duration of a project, although its validation certificate may lapse prior to project completion. This compiler, even if upgraded, must pass all validation tests in force at the time of its validation. A related issue is that of a compiler which is modified to reflect a family of target architectures. These derived compilers are registered and may be considered validated under certain circumstances.

With the expanding supply of validated compilers, attention is focusing on production quality compilers. Validation alone does not ensure good performance, only conformance to the language standard, MIL-STD-1815A. Compilers can achieve efficient object code; the language is not inherently inefficient and does provide many opportunities for compile-time optimization. Furthermore, the programmer may specify some performance improvements directly in the code.

Performance will most likely be affected by run-time checks, generic units, dynamic storage allocation, and rendezvous. These features provide tangible software engineering benefits and although they may entail some run-time overhead, their use should not be restricted. Transformations can be applied selectively at bottlenecks in the program in order to remove inefficiency. Numerous benchmarks are being developed to measure compiler characteristics such as compilation speed, execution speed, and run-time memory requirements.

The run-time support environment is an important component of an Ada system. Interoperability of tools between different environments through standard interfaces is the objective of the CAIS work. Additional research efforts are investigating options in custom tailoring the environment to meet real-time requirements. Existing alternatives include unique run-time environments for each application, smart linking, and run-time customization. Each option affects the development cost, portability, reusability, verification, and reliability.

In order to achieve the transition to software engineering with Ada, program managers must address the issues of training and retraining. Unlike earlier languages, Ada training requires not only language syntax and semantic training but also software engineering and environment training. There is a great deal of language training available in the marketplace. Several full curricula which include methodology and tool training also exist. Both industry and government task forces are studying training issues, including the relative merits of different training media, the time needed to become proficient, and the costs/benefits of certification.

Another aspect of the transition to Ada lies in the use of an Ada program design language. An Ada PDL allows an organization to prepare for Ada by acquiring expertise in the language features which support modern software engineering and design principles. Moreover, when the implementation language is also Ada, overall costs are reduced by choosing a common medium of expression for the two most time-consuming phases of the software development life cycle. In order to gain the full benefits of an Ada PDL, it must be used in conjunction with automated PDL processing tools. At the minimum, such tools would verify the consistency of the design. Sophisticated tools would also facilitate the documentation and coding process.

Finally, in undergoing a transition to Ada, one must consider the transition of the operational software. At one extreme, all of the software would be redesigned and reimplemented in Ada, while at the other end of the spectrum, all of the software would be converted to Ada through an automated translator. Neither extreme is viable. Automated conversion entails many risks, especially the generation of unreadable, unmaintainable code. Converters which apply expert systems technology to the translation process reduce this risk. Complete redesign is usually too costly, both in time and money. A compromise lies in writing enhancements and major overhauls in Ada, using Ada's INTERFACE pragma to link the non-Ada code to the Ada run-time environment, allowing the non-Ada routines to be invoked from the Ada program.

The introduction of Ada into the DoD has resulted in many new issues and complexities which program management must understand. The original Program Manager's Guide to Ada (ESD-TR-85-159) as well as this and upcoming editions of the Program Office Guide to Ada attempt to reflect the progress of Ada by keeping program managers informed of Ada news, issues and their solutions.

LIST OF REFERENCES

[Boo83] Booch, Grady. Object-oriented Development. IEEE Transactions on Software Engineering, SE-12, No. 2 (February 1986), 211-221

[Bra83] Bray, Gary. Implementation Implications of Ada Generics. Ada Letters 3, No. 2 (September-October 1983), 62-71

[F&C86] Flick, R.L., and Connelly, R.W. A Software Development Environment Using PAMELA. Proceedings, First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston, June 1986, D.4.3

[H&N80] Habermann, A.N., and Nassi, I.R. Efficient Implementation of Ada Tasks. Technical report CMU-CS-80-103, Carnegie Mellon University, January 1980

[Hil82] Hilfinger, Paul N. Implementation Strategies for Ada Tasking Idioms. Proceedings of the AdaTEC Conference on Ada, Arlington, Virginia, October 1982, 26-30

[Hoo86] Hood, Philip E. Ada and Cyclic Run-time Scheduling. Proceedings, First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston, June 1986, D.3.3

[KITPR] KAPSE Interface Team, Public Report, Volume I, 1 April 1982, C1

[IEEE86] IEEE Recommended Practice on Ada as a Design Language (preliminary draft), P990 D18, April 2, 1986

[Knu72] Knuth, Donald E. An Empirical Study of FORTRAN Programs. Software -- Practice and Experience 1, No. 2 (April-June 1972), 105-133

[R&M86] Rogers, Patrick, and McKay, Charles W. Implementing distributed Ada for real-time applications. Proceedings, First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston, June 1986, E.3.1

[Wel78] Welsh, J. Economic Range Checks in Pascal. Software -- Practice and Experience 8 (1978), 85-97

BIBLIOGRAPHY

Ada Design Language Developers Matrix, ACM Ada Letters, Vol. IV, No. 3, 1986.

Ada Run-time Environments Working Group (ARTEWG) Report, March 1986, ACM Ada Letters, Vol. IV, No. 3, 1986.

Ardo, Anders and Lars Philipsom, "A Simple Ada Compiler Invalidation test," ACM Ada Letters, April 1984.

Bafes, Paul and Brian West, "Interfacing Ada and other Languages," Proceedings, First International Conference on Ada Programming Language Applications for the NASA Space Station, NASA Johnson Space Center, Houston TX, June 1986.

Chase, A. and M. Gerhardt, "The Case of Full Ada as a Design Language," ACM Ada Letters, Vol. II, No. 3, 1982.

Grover, Vinod, "Guidelines for a Minimal Ada Run-Time Environment," (ESD-TR-85-139), SofTech, MA, 1985. AD A160451

Harbaugh, Sam and John A. Foraris, "Timing Studies Using a Synthetic Whetstone Benchmark," ACM Ada Letters, Vol. IV, No. 2, 1984.

Hood, Philip E. and Vinod Grover, "Real Time Support Issues for an Ada Run-Time System," SofTech, December 1985.

Hook, A. Audrey, Gregory A. Riccardi, Michael Vilot, and Stephen Welke, "User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC) Version 1" (IDA Paper P 1879), Institute for Defense Analysis, VA, October 1985.

Howe, R.G., M. Hagle, W.E. Byrne, E.C. Grund, R.F. Hilliard II, and R.G. Munck, Program Manager's Guide to Ada (ESD-TR-85-159), May 1985.

Kamrad, M. "Run-time Organization for the Ada Language System Programs," ACM Ada Letters, Vol. III, No. 3, 1983.

Laird, James D., Dr. Bruce A. Burton, and Mary R. Koppes, "Implementation of an Ada Real-Time Executive - A Case Study," Proceedings, First International Conference on Ada Programming Language Applications for the NASA Space Station, NASA Johnson Space Center, Houston TX, June 1986.

LeGrand, S. and R. Thall "The CAIS 2 Project," Proceedings, First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston TX, June 1985, D.2.5.

Martin, Donald G., "Non-Ada to Ada Conversion," Journal of Pascal, Ada, & Modula-2, Vol. IV, No. 6, 1985.

Sheffield, James R., "Ada Programming Design Language - A Report on their Status," Naval Avionics Center, Indianapolis IN, 1984.

Santhanam, V. "A Practical Approach for Translating FORTRAN Programs to Ada," Proceedings of the Fourth National Conference on Ada Technology, Fort Monmouth NJ, March 1986.

SofTech, "Ada Programming Design Language Survey, Final Report," Naval Avionics Center, Indianapolis IN, October 1982.

SofTech, "Analysis of Concerns Raised About Space Systems Use of Ada," Prepared for NASA Headquarters Office at Space Station, January 1986.

SofTech, "JAMPS PDL Guide," October 1984.

SRI International, "The Suitability of Ada for Artificial Intelligence Applications," Report for the Army Research Office, May 1980.

TRW Defense System Group, "A Risk Management Approach to CAIS Development," Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston TX, June 1986.

Wallis, P. J. L., "Automatic Language Conversion and its Place in the Transition to Ada," Proceeding of Ada International Conference, Cambridge University Press, 1985.

Wichman, B. A., "Ackermann's Function in Ada," AMC Ada Letters, Vol. IV, No. 3, 1986.

Yavne, Nancy Linden, "A Simple Approach to a Relaxed Syntax for an Ada PDL," ACM Ada Letters, Vol. V, No. 1, 1985.

INDEX

APPENDIX 1

Mission Critical Computer Resource (MCCR)
Focal Points

As of 14 February 1986

AD/ENE
Attn: Ms. Sharon Brooks
Eglin AFB FL 32542-5000
AUTOVON 872-8505
Comm (904) 882-8505

AFWAL/AAAF
Attn: Ms. Donna Morris
Wright-Patterson AFB OH 45433-6543
AUTOVON 785-3826
Comm (513) 255-3826

AFCMD/EPER
Attn: Mr. Brown
Kirtland AFB NM 87117-5000
AUTOVON 244-0859
Comm (505) 844-0859

BMO/ACD
Attn: Lt Col Stajanowski
Norton AFB CA 92409-6468
AUTOVON 876-4620/482
Comm (714) 832-4620/4829

ASD/EN (CRFP)
Attn: Mr. Babel
Wright-Patterson AFB OH 45433-6503
AUTOVON 785-3656/2146
Comm (513) 255-3656/2146

AMD/SIX
Attn: Mr. Muniz
Brooks AFB TX 78235-5000
AUTOVON 240-3264/2369
Comm (512) 536-3264/2369

ESD/ALS
Attn: Mr. Kent
Hanscom AFB MA 01731-5000
AUTOVON 478-5023
Comm (617) 861-5023

AFATL/DLCM (CRFP)
Attn: Ms. Anderson
Eglin AFB FL 32542-5000
AUTOVON 872-2961
Comm (904) 882-2961

RADC/COEE
Attn: Mr. Motto
Griffiss AFB NY 13441-5700
AUTOVON 587-3655
Comm (315) 330-3655

AFALC/AXTS
Attn: Lt Col Murphy
Wright-Patterson AFB OH 45433
AUTOVON 785-5945
Comm (513) 255-5945

SD/ALR
Attn: Lt Col Stevens
P.O. Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960
AUTOVON 833-2532
COMM (213) 643-2532

INFORMATION:

AEDC/SI
Attn: Mr. Bond
Arnold AFS TN 37389
AUTOVON 340-5454
Comm (615) 455-5454

AFSTC/XNR
Attn: Captain Strickland
Kirtland AFB NM 87117
AUTOVON 246-5545
Comm (505) 846-5545

AFFTC/SI
Attn: Mr. Vonklargaard
Edwards AFB Ca 93523-5000
AUTOVON 350-2344
Comm (805) 277-2344

6575 School Squadron
Attn: Captain Vinyard
Brooks AFB TX 78235-5000
AUTOVON 240-2770
Comm (512) 536-2770

ESMC/RSC
Attn: Mr. Thorne
Patrick AFB FL 32925
AUTOVON 854-2001
Comm (305) 494-2001

AFALC/EREC
Attn: Captain Carlton
Wright-Patterson AFB OH 45433-5000
AUTOVON 785-4991
Comm (513) 255-4991

WSMC/EN
Attn: Mr. Salazar
Vandenberg AFB CA 93437-6021
AUTOVON 276-7968
Comm (805) 866-7968

Det 1, HQ AFSC/IGK
Eglin AFB FL 32542

APPENDIX 2

THE UNDER SECRETARY OF DEFENSE

WASHINGTON D.C 20301

RESEARCH AND
ENGINEERING

2 DEC 1985

MEMORANDUM FOR SECRETARIES OF THE MILITARY DEPARTMENTS
CHAIRMAN OF THE JOINT CHIEFS OF STAFF
UNDER SECRETARY OF DEFENSE FOR POLICY
ASSISTANT SECRETARY OF DEFENSE, COMPTROLLER
ASSISTANT SECRETARY OF DEFENSE, ACQUISITION AND LOGISTICS
ASSISTANT SECRETARY OF DEFENSE, COMMAND, CONTROL, COMMUNICATIONS AND INTELLIGENCE
GENERAL COUNSEL, DEPARTMENT OF DEFENSE
INSPECTOR GENERAL, DEPARTMENT OF DEFENSE
DIRECTOR, DEFENSE INTELLIGENCE AGENCY
DIRECTOR, DEFENSE NUCLEAR AGENCY
DIRECTOR, NATIONAL SECURITY AGENCY
DIRECTOR, DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

SUBJECT: Implementation of Ada* in Department of Defense Programs

Since June 1983, when we stated our intention to establish Ada as the single, common computer programming language for Defense mission critical applications, the Military Departments and agencies have initiated numerous efforts to facilitate that process.

We must continue to enforce the use of Ada in new systems and seek opportunities to insert Ada technology into major software system upgrades in order to reap the benefits of increased productivity and reliability. Based upon early examples of successful use of the Ada language and the stability of Ada compilers, the time has come to capitalize on the investment which has been made in Ada by the department, various government agencies, and industry. Your support of this important initiative to improve defense software is appreciated.

Donald A. Hicks

*Ada is a registered trademark of the U. S. Government (Ada Joint Program Office).

APPENDIX 3

VALIDATION POLICIES AND PROCEDURES

PART I - VALIDATION POLICY

PART II - THE USE OF ADA* COMPILERS IN DoD

PART III - PROCEDURES FOR CONDUCT OF THE ADA* VALIDATION PROCESS
(NOT INCLUDED)

DRAFT

24 January 1986

* Ada is a registered trademark of the U. S. Government (Ada Joint Program Office).

PART I

VALIDATION POLICY

1.0 PURPOSE. Validation of an Ada compiler is the process of testing the conformity of the compiler to the Ada programming language standard, ANSI/MIL-STD-1815A. The goal of validation is to prevent the proliferation of subsets, supersets, or dialects of the Ada language, in order to promote software re-useability and to reduce life-cycle costs.

This validation policy provides the means to realize the goal of validation, while minimizing the hindrances for the availability of Ada compilers that may be caused by the requirement of validation.

2.0 SCOPE: This document defines the general framework for the process of Ada Compiler validation, which is responsive to both general trade and DoD concerns, and to identify the reponsibilities of all parties directly involved in the validation process. A policy regarding DoD-specific procurement and project management issues that are related to the use of Ada compilers, rather than to the validation process itself, is separately formulated in "PART II - THE USE OF ADA COMPILERS IN DoD." Detailed procedures for implementing this validation policy are specified in "PART III - PROCEDURES FOR CONDUCT OF THE ADA VALIDATION PROCESS."

3.0 DEFINITION OF TERMS:

Ada Compiler: The compilation and execution system required to compile and execute Ada programs in accordance with the Ada language standard, ANSI/MIL-STD-1815A.

Ada Compiler Validation Capability (ACVC): The set of Ada programs that test the conformity of a compilation and execution system to the Ada language standard, ANSI/MIL-STD-1815A, in addition to the documentation and tools that facilitate the conformity testing.

Base Compiler: The Ada Compiler originally tested as part of the validation process.

Base Configuration: The host machine, host operating system, target architecture, and target operating system (if any) under which the Base Compiler is originally tested as part of the validation process.

Derived Compiler: A Base Compiler that has been modified for any reason, or a Base Compiler in a configuration not fully tested by an AVF, which is affirmed by the vendor to remain completely in conformity to the Ada language standard, ANSI/MIL-STD-1815A.

Validated Compiler: A Base Compiler for which a Validation Certificate is in effect, a Derived Compiler that has been registered with the AJPO, and any versions of these compilers maintained in conformity with the Ada language standard. A Base Compiler and any derivation of that compiler will be considered validated compilers while the Base Compiler's Validation Certificate is in effect.

Validation: The process of checking the conformity of an Ada compiler to the Ada Standard, ANSI/MIL-STD-1815A.

Validation Certificate: The certificate issued by the Ada Joint Program Office that certifies the successful test of a Base Compiler on a Base Configuration against all ACVC tests that are applicable for the specified Base Compiler and Base Configuration.

Vendor: The supplier of an Ada compiler.

4.0 VALIDATION OF BASE COMPILERS: Ada Compilers shall be validated and subsequently revalidated on a periodic basis. A successful validation shall consist of a successful test of the Ada Compiler against all applicable tests provided by a version of the ACVC that is admissible for validation at the time of testing. The testing of the Ada Compiler as part of validation shall be performed by an independent team which operates under the auspices of an Ada Validation Facility (AVF) as authorized by the Director of the AJPO. The results of such testing shall be documented in a Validation Summary Report (VSR).

After successful completion of the testing of an Ada Compiler and preparation of the VSR, a Validation Certificate shall be issued by the Director of the AJPO to the Vendor. The Validation Certificate shall uniquely identify the Base Compiler and Base Configuration as well as the version of the ACVC under which the testing was performed.

The period after which Validation Certificates expire shall be determined by the Director of the AJPO. An automatic extension, valid until adjudication of a pending revalidation, shall occur whenever a Vendor has submitted his Compiler for revalidation in a timely manner as defined by the validation procedures. In such cases the existing Validation Certificate shall be considered valid until the VSR for the revalidation has been issued. Compilers for which current Validation Certificates exist shall be considered to be Validated Compilers.

The AJPO will maintain and make publicly available a list of Ada Compilers for which Validation Certificates have been issued. The AJPO will also make the VSR publicly available for any compiler on this list.

5.0 REGISTRATION OF DERIVED COMPILERS: In accordance with validation procedures, Vendors may submit requests for the registration of Derived Compilers with the AJPO following the successful validation of a Base

Compiler. As part of this registration, the Vendor shall affirm that the Derived Compiler is a correct implementation of the Base Compiler on a configuration other than the Base Configuration and that the Derived Compiler conforms to the Ada language standard, ANSI/MIL-STD 1815A. The AJPO may request additional information to be provided by the Vendor to credibly substantiate the claim of conformity to the standard.

The AJPO will maintain and make publicly available a list of all registered Derived Compilers, together with a description of their configurations, of their relation to a Base Compiler, and of any information supplied by the Vendor in substantiation of compliance to the standard. No Validation Certificate will be issued for a Derived Compiler.

Registered Derived Compilers shall be considered as Validated Compilers. This status expires no later than the Validation Certificate of the associated Base Compiler. A Derived Compiler will be removed by the AJPO from the list of Registered Derived Compilers and no longer considered a Validated Compiler, if it is determined to fail an applicable ACVC test.

6.0 MAINTENANCE OF VALIDATED COMPILERS: Maintenance changes to a Validated Compiler do not affect its status as a Validated Compiler, provided that the compiler continues to be in conformity to the language standard. A maintained compiler shall not be advertised as a Validated Ada Compiler if it is known that this compiler fails an applicable ACVC test that was passed by the associated Base Compiler during the validation testing.

The nomenclature of the version identification of a maintained Base Compiler shall differ from the version identification of the Base Compiler noted on the Validation Certificate for any Ada Compiler that includes maintenance changes.

7.0 RESPONSIBILITIES:

The Director of the Ada Joint Program Office (AJPO) shall:

o Be responsible for establishing and maintaining the Ada Validation Process

o Designate an Ada Validation Organization (AVO) and delegate the authority for managing the Ada Validation Process to the AVO

o Approve the establishment of Ada Validation Facilities (AVFs) to perform the actual validations according to this validation policy and the validation procedures established for and by the AVO

o Have final authority in the decision over disputes raised by Vendors over validation issues.

The AVO shall:

- Establish a detailed set of guidelines and procedures which is consistent with the validation policy and procedures set by the AJPO. These guidelines and procedures, which shall be approved by the Director of the AJPO, shall establish the operating guidelines for the AVO and AVFs.

- Ensure that the validation policy and the established guidelines and procedures are consistently followed by all AVFs

- Maintain accurate records pertaining to each validation and to the validation process

- Maintain the list of Validated Compilers

- Advise the Director of the AJPO on all validation issues.

Each AVF shall follow the guidelines and procedures set forth by the AJPO and AVO. The AVFs shall be responsible for conducting validation in a timely and impartial manner, for producing the Validation Summary Report (VSR), and for forwarding disputes raised by a Vendor to the AVO for a binding decision.

Vendors are responsible for providing accurate and sufficient information to perform the validation process or to register a Derived Compiler according to the established policies and procedures. Vendors are responsible for maintaining the conformity of their Ada Compilers to the Ada language standard, ANSI/MIL-STD 1815A.

Users are responsible for understanding the scope and limitations of compiler validation, which is a means to increase confidence in the conformity of an Ada compiler to the Ada language standard. While such conformity is a first measure of usability of the compiler, it by no means guarantees that a Validated Compiler satisfies all usability requirements of a particular project.

PART II

THE USE OF ADA COMPILERS IN DoD

1.0 PURPOSE. The purpose of this document is to provide policy and applicable guidelines to promote the use of Ada for Mission Critical Computer Resource (MCCR) programs.

2.0 SCOPE. This document integrates the policies and procedures of Ada compiler validation (references a and b) with the need for developing, deploying, and maintaining MCCR software in accordance with DoD life-cycle management policy, procedures, and practice, using Ada compilers that conform to the Ada language standard, ANSI/MIL-STD-1815A. This policy applies to all managers of DoD MCCR programs and provides guidance to these managers in their compliance with the required use of Ada compilers that conform to the Ada language standard.

3.0 DEFINITION OF TERMS: The following terms are used in the DoD General Policy for Validation and associated procedural guidelines for implementing that policy. List I repeats the terms that apply to general use of Ada compilers (including MCCR projects) provided in Reference (a). List II defines the terms that specifically apply to MCCR projects.

List I: General Terms

Ada Compiler: The compilation and execution system required to compile and execute Ada programs in accordance with the Ada language standard, ANSI/MIL-STD-1815A.

Ada Compiler Validation Capability (ACVC): The set of Ada programs that test the conformity of a compilation and execution system to the Ada language standard, ANSI/MIL-STD-1815A, in addition to the documentation and tools that facilitate the conformity testing.

Base Compiler: The Ada Compiler originally tested as part of the validation process.

Base Configuration: The host machine, host operating system, target architecture, and target operating system (if any) under which the Base Compiler is originally teZted as part of the validation process.

Derived Compiler: A Base Compiler that has been modified for any reason, or a Base Compiler in a configuration not fully tested by an AVF, which is affirmed by the vendor to remain completely in conformity to the Ada language Standard, ANSI/MIL-STD-1815A.

Validated Compiler: A Base Compiler for which a Validation Certificate is in effect, a Derived Compiler that has been registered with the AJPO, and any versions of these compilers maintained in conformity with the Ada language standard. A Base Compiler and any derivation of that compiler will be considered validated compilers while the Base Compiler's Validation Certificate is in effect.

Validation: The process of checking the conformity of an Ada compiler to the Ada Standard, ANSI/MIL-STD-1815A.

Validation Certificate: The certificate issued by the Ada Joint Program Office that certifies the successful test of a Base Compiler on a Base Configuration against all ACVC tests that are applicable for the specified Base Compiler and Base Configuration.

Vendor: The supplier of an Ada compiler.

List II - MCCR Program Terms

Generic Target: A hardware and/or software implementation of a Real MCCR Target that is equivalent to or a superset of the real target and is capable of executing all applicable ACVC tests. A Generic Target ;s equivalent to the Real MCCR Target if it possesses the same instruction set and run-time interface. A superset of a Real MCCR Target is one to which the Real MCCR Target could be made equivalent by adding more memory, input-output capabilities, instructions, etc.

Program Manager: An individual who has responsibility and accountability for the acquisition and/or maintenance of a DoD system.

Project-Validated Compiler: A validated Ada compiler which is baselined in accordance with DoD life-cycle management policies, procedures, and practices. Such a compiler retains its status as a Project-Validated Compiler throughout the duration of the project: its status as a Validated Compiler is not retained beyond one year.

Real MCCR Target: A hardware component of an MCCR system that has been designed to comply with operational form, fit, and function specifications of the MCCR system which may execute object code generated by an Ada

compiler.

Restricted Target: A Real MCCR Target on which not all ACVC tests can be executed but which can execute object code generated by a validated compiler and an application specific run-time library.

4.0 POLICY. The following policy applies to the management of projects that develop or maintain Ada software for use in MCCR programs.

4.1 USE OF PROJECT-VALIDATED ADA COMPILERS: Any MCCR Ada software delivered for operational testing, deployment and maintenance shall be compiled with Project-Validated Compilers. A Project-Validated Compiler shall maintain its status for the duration of a project and any contractual arrangement that requires usage of this Project-Validated Compiler, regardless of the validation status of the compiler under the general validation policies and procedures. Maintenance of Project-Validated Compilers shall not affect its status as a Project-Validated Compiler, as long as the modified Compiler is capable of passing all applicable tests of the ACVC in a version equal to or more recent than the ACVC version that was in effect at the time of the baselining of the Project-Validated Compiler.

4.2 ADA COMPILERS FOR RESTRICTED TARGETS: An Ada compiler used to generate object code for a Restricted Target will be considered to be a Project-Validated Compiler if all of the following conditions are satisfied.

a. The compiler was derived from a Project-Validated Compiler for a Generic Target.

b. The Project-Validated Compiler for the Generic Target is a fully conforming implementation of the Ada language, even though its use may be solely for the development of application software for Restricted Targets.

c. All mandatory features of the Ada language that can be supported on the Restricted Target are supported by the compiler for the Restricted Target. Compilers for the Restricted Target shall not be arbitrarily constrained to sub-set implementations of the Ada language.

d. The code executes on the Restricted Target in confomance with the Ada language Standard.

e. All application-specific run-time libraries for Restricted Targets shall be contained within that application and shall not affect the Ada compiler for

the Generic Target or the Restricted Target when used to generate code for other applications.

4.3 ADA COMPILERS IN MCCR SOFTWARE DEVELOPMENT: Ada Compilers used to develop MCCR software are not required to be Project-Validated throughout the development phase. The Program Manager shall determine when the compilers shall be baselined for development of that software, at which time the compiler must satisfy the criteria of a Project-Validated Compiler.

4.4 QUALITY ASSURANCE DURING OPERATIONAL TESTING: Upon delivery of a MCCR software release for operational testing, the Project-Validated Compiler used to compile this software shall be tested against all applicable tests of the ACVC in a version equal to or more recent than the ACVC version that was in effect at the time of baselining the compiler. Quality assurance testing of the Project-Validated Compiler shall be implemented in accordance with the action that is appropriate for the size of the software release: The following criteria apply:

a. Less than 2,000 Ada statements:
Compiler testing is optional.

b. Up to 100,000 Ada statements:
Compiler testing by the Program Manager is mandatory.

c. Over 100,000 Ada statements:
Validate according to the policies for general Validation.

When an MCCR software release has been compiled on several Project-Validated Compilers, these acceptance testing requirements apply to each of these compilers and the respective Ada source code compiled with them. The requirement for acceptance testing of a Project-Validated Compiler shall be waived if the Project-Validated Compiler is an unmodified Base Compiler for which a Validation Certificate is in effect.

4.5 QUALITY ASSURANCE DURING DEPLOYMENT AND MAINTENANCE: At each baseline milestone in the maintenance cycle, the acceptance testing of a Project-Validated Compiler shall be repeated as specified for acceptance during operational testing. These requirements are waived if the Project-Validated Compiler is identical to the one of the previous baseline milestone, or if it has been replaced by an unmodified Base Compiler for which a Validation Certificate is in effect.

4.6 MAJOR SYSTEM UPGRADES: Major system upgrades shall be

combined with an upgrade of the Project-Validated Compilers to the Validated Compilers. The results of acceptance testing of these Validated Compilers shall be part of the project documentation.

5.0 PROCEDURES. These procedures provide guidance to program managers in implementing policy for the initial acquisition of an Ada compiler, use of that compiler through the software development phase, the transition from development to maintenance activity, and maintenance activity.

5.1 INITIAL ACQUISITION OF AN ADA COMPILER FOR AN MCCR PROJECT: A program manager is responsible for identifing the requirement for the delivery of a validated Ada compiler as an action within the context of project milestones. A compiler may be selected from the registered list of Derived Compilers or may be a Base Compiler with a current Validation Certificate. It is recommended that acquisition of all Validated Compilers for MCCR software development or maintenance be contingent on a successful testing against all applicable ACVC tests. This condition is automatically satisfied by selection of an unmodified Base Compiler for which a Validation Certificate is in effect. If the compiler is developed in-house, the program manager will be required to obtain a certificate for this compiler in accordance with the formal validation process explained in reference c.

5.2 SOFTWARE DEVELOPMENT WITH A VALIDATED ADA COMPILER: Ada software may be developed prior to obtaining a Validated Compiler and baselining this compiler as a Project-Validated compiler. However, use of a Validated or Project-Validated Compiler at the earliest practical time is encouraged, to reduce risk and potential problems during the acceptance of the software for operational testing. When a validated compiler has been accepted for a project, configuration control procedures should be established to ensure complete documentation for changes made to the Project-Validated Compiler and for derivations from it. Program managers are encouraged to ascertain in periodic intervals that maintenance changes and derivations have not affected the capability of the Project-Validated Compiler to pass all applicable ACVC tests used to initially validate it. Program managers are encouraged to update the Project-Validated ompilers for their projects at major project milestones. After expiration of a validation certificate for a Project-Validated compiler, a program manager will ensure that the compiler is a conforming implementation throughout the life of the project by taking the following actions:

a. Re-test the Project-Validated Compiler and derived compilers using the ACVC version used to originally establish the conformity of the base compiler. This periodic re-testing may be scheduled as part of project baseline milestones. A program manager will determine

whether this testing will be done by project personnel or by an AVF. Cost, schedules, and contractual obligations will be considerations in determining the conduct of re-testing.

b. At each baseline milestone, certify that the Project-Validated Compiler has successfuly passed all applicable ACVC test. This certification will become part of the project documentation .

c. Ensure that all application specific run-time libraries for Restricted Targets are developed and documented as modules of the application software, and that these libraries do not affect the validatable status of the Ada compiler used to generate object code.

d. Ensure that planned program product improvement (P3I) actions are incorporated into project baseline milestones and contracts well in advance of the projected action. P3I actions may result in the acquisition of a replacement compiler with a current validation certificate. Total project cost and schedules will be considerations for P3I actions which must be approved by a program manager.

5.3 TRANSITION TO MAINTENANCE: A program manager is responsible for defining the test and evaluation requirements for the host/target configurations and support software that will be required for the maintenance activity. The program manager will be responsible for ensuring that, to the maximum practical extent, the inventory of application specific compilers are minimized. The use of a Generic Target which is capable of supporting multiple targets is strongly encouraged. The acceptability of Ada applications for maintenance activity will be contingent upon the program manager's compliance with baseline ACVC testing to establish the conforming status of Project-Validated Compilers and operational suitability testing delineated by the maintenance activity.

5.4 MAINTENANCE: Since MCCR software generally has a long operational period, P3I actions will be required for applications and for the software support environment. Upgrades for the software support environment will be maintained by periodic replacement of Ada compilers that are formally validated and will be baselined and periodically re-tested according to the procedures used during software development.

6.0 ORGANIZATIONAL ROLES AND RESPONSIBILITIES:

a. The Ada Joint Program Office (AJPO) shall:

o Maintain DoDproject case histories as provided by DoD program managers for specific projects and make these

available to all DoD program managers to facilitate the use of Ada.

o Provide technical and policy guidance to program managers with respect to specific issues.

o Update guidelines to reflect experience of program managers.

b. Program Management Organizations shall:

o Provide program guidance on the use of Ada.

o Maintain configuration control procedures for the use of Ada.

o Provide procurement guidance that clarifies the requirements for a Project-Validated compiler, its testing, and P3I actions.

o Establish a project reporting system to track the implementation of this guidance.

o Maintain laison with the AJPO so that issues can be resolved at the earlist possible time.

7.0 REFERENCES.

a. Part I - Validation Policy: Validation Policies and Procedures. Draft of 24 January 1986

b. Part II - Use of Ada Compilers in DoD: Validation Policies and Procedures. Draft of 24 January 1986.

c. Part III- Procedures for Conduct of the Ada Validation Process: Validation Policies and Procedures. Draft of 24 January 1986.

APPENDIX 4

Points of Contact for Ada Information

Ada Joint Program Office Points of Contact

Virginia Castor Director	Ada Joint Program Office 3D139 (1211 S. Fern, Rm. C-107) The Pentagon Washington, DC 20301-30812 (202) 694-0210
LTC David Taylor	Army Deputy Director of AJPO Ada Education & Training (including ASEET Team and AFCEA study) International & NATO Ada activities
LCDR Philip Myers	Navy Deputy Director of AJPO Ada Environments (including KIT, KITIA, E&V and SIGAda ARTEWG effort)
Maj. Allan Kopp	Air Force Deputy Director of AJPO Ada Promotion, Ada Trademark
Ray Boswell	Advisor to AJPO
Burt Newlin	Standardization

DoD Ada Related Programs and Points of Contact

Ada Information Clearinghouse	3D139 (1211 S. Fern, Rm. C-107) The Pentagon Washington, DC 20301-3081 (703) 685-1477
Ada Integrated Environment (AIE)	Elizabeth Kean RADC/COEE Griffiss AFB, NY 13441 (315) 330-2762
Ada Language System (ALS)	United States Army Communications Electronics Command (CECOM) Ft. Monmouth, NJ 07703 POC: Dennis Turner (201) 544-4149

Ada Math Library	Naval Ocean Systems Center Code 423 San Diego, CA 92152 POC: Gil Myers (619) 225-7401
Ada Validation Office	Audrey Hook Institute for Defense Analyses 1801 Beauregard Street Alexandria, VA 22311 (703) 845-5501
Ada Verification Technology	Terry Mayfield Institute for Defense Analyses 1801 Beauregard Street Alexandria, VA 22311 (703) 845-2263
Ada Compiler Validation Capability (ACVC)	Language Control Facility (LCF) ASD/SIOL Wright-Patterson AFB, OH 45433 POC: Georgeanne Chitwood (513) 255-3813
DIANA	Commanding Officer Naval Research Laboratory Attn: Code 5150 POC: Rudy Krutar (202) 767-2197
E&V Team	AFWAL/AAAF Wright-Patterson AFB, OH 45433 POC: Ray Szymanski (513) 255-2446
CAIS	Naval Ocean Systems Center 421 Catalina Boulevard San Diego, CA 92152 POC: Patricia Oberndorf (619) 225-6682
United States Air Force Program Manager	Col. Kenneth Nidiffer HQ AFSC/PLR Andrews AFB, MD 20331 (301) 981-5731
United States Army Program Manager	Col. Harold Archibald AMCDE-SB 5001 Eisenhower Avenue Alexandria, VA 22333 (703) 274-9310

United States Navy
Program Manager

CDR Scott Gordon
Space & Naval Warfare Systems
Command, SPAWAR 034
Washington, DC 20363-5100
(202) 692-3966

Ada Language Journals and Newsletters

Ada Information Clearinghouse Newsletter
3D139 (1211 S. Fern, Rm. C-107)
The Pentagon
Washington, DC 20301-3081
(703) 685-1477

Ada-Jovial Newsletter
ASD/SIOL
Computer Operations Division
Information Systems & Technology Center
Wright-Patterson AFB, OH 45433-6503
(513) 255-4472/4473

Ada Letters
Association for Computing Machinery
11 West 42nd Street
New York, NY 10036
(212) 869-7440

Journal of Pascal, Ada & Modula-2
Wiley Journals
John Wiley & Sons, Inc.
605 Third Avenue
New York, NY 10158

Point of Contact for Some Ada Programs and Activities

ABICS Ada Based Integrated Control System	Paul Korkemaz McDonnell Douglas (314) 234-3623	
CAMP Common Ada Missile Package	Ms. Chris Andersen AFATL/DLCM Eglin AFB, FL 32542-5000 AV 872-2961 (904) 882-2961	
MILSTAR	Lt. Col. Kacena (SYSTO) AV 858-6885 (301) 981-6885	Col. McNevin (PM) AV 833-1834 (213) 643-1834
MIMS Mobile Information Management System	Maj. Smith HQ SAC/SICA Offutt AFB, NB (402) 294-3412	
SRAM II Short Range Attack Missile	Cpt. Bauman AV 858-3356 (301) 981-3356	Col. Bevelhymer AV 785-5080 (513) 255-5080
SARAH Standard Automated Remote Autodyn Host	Cpt. Salisbury CCSO/SKAS Tinker AFB, Oklahoma City OK 73145 (405) 734-2457	
WIS World Wide Military Command and Control Information System	Ltc. Courtwright (703) 285-5065	
Missile Warning System	Col. Egolf HQ, AFSpaceCOM/LKD5T Peterson AFB, CO 80914-5001	
ARTEWG Ada Run-Time Environment Working Group	Mike Kamrad Honeywell, M/S MN65-2100 3600 Marshall St. NE Minneapolis, MN 55418 (612) 782-7321 Kamrad @ HI-MULTICS	
ASEET Ada and Software Engineering Education and Training	Maj. Samuels AV 868-3728 (601) 377-3728	

Points of Contact for Ada Programs and Activities

Keesler AFB	Mary Rivers AV 868-3110 (601) 377-3110
AFIT	Lt. Col. Rick Gross AV 785-3098 (513) 255-3098
USAFA	Maj. Nielson AV 259-4112 (303) 472-3590
MCS	Bob Whited Col. Brooks LaGree Chief of Software Support Division Maneuver Control Software Support Division, Software Life Cycle Engineering Center Ft. Leavenworth, KS (913) 684-7642

An updated list of AFSARC/DSARC Ada programs and points of contact may be found in a future Edition of this Guide